3''

/A THOUGHT EXPERIMENT TO DETERMINE THE KNOWLEDGE REQUIREMENTS OF AN EXPERT SYSTEM TO ANALYZE YOURDON - CONSTANTINE DESIGN HIERARCHIES/

by

RICHARD E. COURTNEY

B. S., Southwestern College, 1976

A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1986

Approved by:

David A. Gustafson

Major Professor

TABLE OF CONTENTS

TABLE OF FIGURES

Chapter 1

Introduction

"The purpose of this module is intuitively obvious to even the most casual of observers."

Anonymous

1.1 Project Design

The purpose of this work is the determination of the information needed for an expert system so it can analyze a software design document. The goal is for the expert system to examine an Entity - Relationship - Level (ERL) design specification [Gu84] and to evaluate it for completeness, workability and understandability. To do this evaluation the expert system must extract the appropriate information from the ERL and must possess the requisite knowledge to do the analysis.

The ERL provides information about entities, and relationships between entities. The ERL is a frame-based document that has frames for activities, data types and control information. The ERL document [Figure 2.2] is a product of the requirement phase of the software life cycle designed to provide the information base required to support software development and maintenance through the entire life cycle. The ERL at the end of the requirements phase provides a description of inputs, outputs, activities and nodes. Complex data structures can be described in terms of simpler data items.

When the project moves to the design stage the ERL continues to be used because it allows for the addition of activities as entities with relations to the data. As more details about the project are determined, more slots are filled in this document, including control information via call relationships. A hierarchy diagram for the design can be created from information in the ERL at the end of the preliminary design phase.

From the ERL document the system will obtain the names of the inputs to a module, the data types of the inputs, and the names and structures of the outputs. The system can calculate where the inputs belong in the data flow. The ERL provides control information with the "calls" slot. The "keyword" slot describes the general purpose of the module. The system can combine this information with the name of the module to give clues about the function of the module. One additional field introduced into the ERL is an intrinsic knowledge field that will describe information to be embedded in the module code.

The data structure of the inputs and outputs of the module represents extremely valuable information. Many data structure texts [Ah74] [Tr84] and other books [Wi76] [De84] define what operations are allowed on each of the various data structures. A major aspect of the expert system's design analysis will be to make a correspondence between the data type and the operations proposed. The combination of the keyword of a module and the module name will assist the system to determine the primitive type or combinations of primitive types of operations the module is designed to do. Experience and experiments have shown that people choose meaningful names for their modules [So84] [Do84]. The expert system will scan the name for such meaningful words.

The ERL supports top down design, which fits well with maintaining abstraction levels of both design and data structures. An aspect of the design analysis will be to compare the given control flow with the analyzed data flow to present the designer with alternative designs and to reveal potential concurrency in the system.

The intrinsic knowledge field, to be added to the ERL, allows this design document to pass along the same information as the requirement documents. Many cases exist where data are encoded in the module code, but could be perceived as input. For example, a module that has an error message as output must have some data to make a comparison with to determine an out-of-domain error. These data could either be entered as parameters or written straight into the code. The data are known to exist because of the type of operations being performed, and should be added to the ERL in the intrinsic knowledge slot.

The Yourdon - Constantine approach to designing software projects used in this paper is called structural design [Yo79]. Modules, programs or subsystems are identified and appropriate relationships are established between the activities. In other words, a person using this method designs major components of the system and then works on the details of the individual components.

1.2 Background Information

The development of an expert system to analyze designs is possible at this time due to the recent work being done at several locations. The design analysis has foundations in the work of Charles Rich at MIT on the Programmer's Apprentice [Ri81]. His work in developing program plans and the work by Elliot Soloway et. al. at Yale [Ad85] and a group at Advanced Information and Decision Systems including Eric Domeshek, Brian McCune and Jeffrey Dean demonstrate that programmers have generalized, stereotyped algorithms to design systems with, and a relative standard vocabulary that they use to reference these concepts. The impetus of this project compared to the works mentioned above is to move the automated analysis one level forward in the software life cycle. The expert system is to analyze the product of the detailed design phase which is still at an abstract level. Code transformations may be viewed as an automated task, i.e. simply compiling higher level languages, whereas design is still viewed as an art form. Code transformation systems are programmed to proceed depth-first into the problem. The design is assumed to be correct so that the transformation need only produce the correct code for the operation at hand. For contrast, the person analyzing a design is concerned with assuring that the different pieces fit together in the correct manner.

Showing how an expert system would go about analyzing the design will help formalize rules as to what makes a good design. In this effort I will attempt to formalize the design rules and will show how they will work on different designs.

1.3 Hypothesis

This thesis proposes that:
The rules to analyze software design for correctness given the limited information of module names, inputs, outputs, calls, and keywords from

and Entity - Requirement - Level document can be formalized for manual application.

In this manner the rules can be made precise enough to form the basis on which the expert system can rest.

People have many types of expectations, such as a "How are you?" after greeting a friend on the street. An example of a design expectation would be that an output file implies an input file. That is, if there is a module in the system that outputs a file to secondary storage then there should also be a module to input a file from secondary storage. Another expectation would be if two different software designers were given the same set of requirements, two different designs would be produced. However, one could expect that certain features would exist in the design because they exist in almost all software designs. A design which is closer to these preconceived ideas is easier to understand and therefore less likely to have errors introduced and easier for people to maintain.

The standard pieces the expert system will look for include modules to input data and modules to output data. The system expects to see afferent, transform, efferent, and coordinate sections. The afferent modules are responsible for accepting and massaging data into the system. This includes any error checking and the building of data structures. Modules that take the data and convert them into output form are considered efferent modules. Transform is the label for the pieces in the middle that convert the data. Coordinate modules do the control and switching. Coordinate modules are standard pieces of design as are aggregate data types.

Designing software systems is not an exact science, and as such a debate as to what makes one design better than another design is not absolute. I define a good design of a software system to be a design that in addition to low coupling and high cohesion is consistent in the level of data abstraction input and output from modules across a level of modules. The modules towards the top of the design hierarchy should not introduce details that should be handled at lower levels in the design.

1.4 Guide to Reading

The following chapters provide detail on how the rules for automated analysis of a design document were generated. Chapter 2 presents an example of a design and shows what rules are used to evaluate it. Chapter 3 gives a discussion of the developed rules. Chapter 4 presents a hierarchy of data types and the operations that may be performed on the structures. Chapter 5 demonstrates the consistency of the rules by applying them to another design problem. Chapter 6 is a summary of results and conclusion. Chapter 7 contains a discussion of further research.

Chapter 2

An Unknown Example

2.1 Introduction

To develop the rules used to analyze software design for correctness, I have gone through an example design, extracting the facts I used to reach conclusions about the design. Rules were developed so that if the same facts are present in any design, the same conclusions can be reached. I had only the hierarchy diagram and the ERL specification for this analysis. I did not have any other documentation on this program, nor was I familiar with any portion of the project. The expert system would have only the textual description and not the picture which is very helpful for the human.

2.2 The Design Model

I used the Yourdon - Constantine structured design [Yo79] as a model. The features of this design include a hierarchical structure, and modules are identified as coordinate, afferent, efferent, transform or undetermined. Studies show that one can establish the type of system the design is modeling depending upon the shape of the design.

The hierarchical structure reveals information about individual modules. The higher up in the hierarchy a module is, the more probable it will be that the module is a coordinate module. One can also anticipate that the inputs and outputs will be more complex structures at the higher levels. The modules at the lower levels are more likely to be transform modules.

Modules are identified as coordinate if their function is to control the modules below it. Afferent modules introduce data into the system. Efferent modules work witb the information in the system and pass it to subordinates and out of the system. Afferent and efferent modules may do some manipulation of the data, but modules whose purpose is to take data and perform some type of computation or conversion are designated transform modules. If the purpose of a module cannot be determined, the activity is marked as undetermined.

Although Yourdon and Constantine realize that little can be determined about a design just from the depth or width of the hierarchy, good designs tend to have a definite overall shape. Tbe closest visualization of this shape is a mosque. Tbe concept that underlies the shape is that there is considerable fan-out at the upper level modules and fan-in at the lower levels of the bierarchy.

The shape of the hierarchy also may reveal the type of project the specification represents. If the depth of the afferent section is noticeably greater than the depth of the transform or efferent sections, then the system is input intensive. The converse applies for a large efferent section, it being output intensive. If the system appears to split an input data stream into several separate output streams then the system is trying to model a transaction system over a transformational system. A transform system takes input and proceeds to produce the same type of output each time it is activated. A transaction system generates different types of output besed upon the transaction the user calls.

By taking a software system design specified in the ERL language, a person can try to fit that design to this model and get some ideas as to the type of problem being solved and the sensibility of the design. I will illustrate the knowledge used in a manner that will make it apparent that this knowledge can be coded for machine use.

Once a determination is made about the purpose of a module and the types of input and outputs then expectations are developed about the information that will be needed to perform the perceived operation. If the actual inputs and outputs do not match the presumed data needed, a remark concerning the discrepancy will be made.

2.3 Evaluation

This example was selected from an assignment given to the Software Engineering Project class at Kansas State University in Spring 1986. The class is composed of juniors and seniors majoring in Computer Science and Information Systems, who work in teams on a modest size software project. A typical project will have 1.5 to 2 K lines of code. The unmodified hierarchy diagram is shown in Figure 2.1 and the accompanying ERL specification is shown in Figure 2.2. This team did not use the keyword or mode slots of the ERL which provide more information about the system.

2.3.1 Main module

Analysis of the design starts at the top of the hierarchy chart, with the Main module. Most designs will have this and at least one more layer at the top of the hierarchy which consist of coordinate modules. Main is defined in Webster as "the chief part", thus the module name does lend some information. The other details of the module are that all the outputs are identical to the inputs to the module. In the ERL, if data are used in two different modules, they must be declared as data for all the activities that make a path between the modules. I have determined Main to be a coordinate

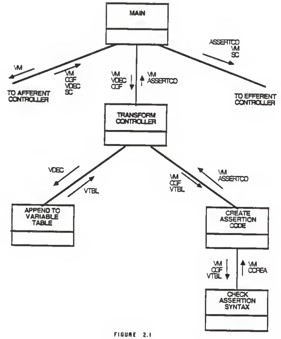
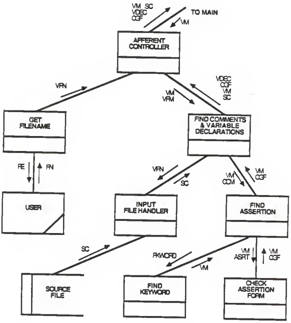

MAIN
ASSERTCD
VM
SC
VM
VM
CGF
VDEC
SC
TO AFFERENT CONTROLLER
VM
VDEC
CGF
VM
ASSERTCD
TO EFFERENT CONTROLLER
TRANSFORM CONTROLLER
VDEC
VTBL
VM
CGF
VTBL
VM
ASSERTCD
APPEND TO VARIABLE TABLE
CREATE ASSERTION CODE
VM
CGF
VTBL
VM
CCREA
CHECK ASSERTION SYNTAX

FIGURE 2.1



VM SC
VDEC
CGF
TO MAIN
VM
AFFERENT CONTROLLER
VFN
VDEC
CGF
VM
SC
VM
VFM
GET FILENAME
FIND COMMENTS & VARIABLE DECLARATIONS
FE
FN
USER
VFN
SC
VM
CCM
VM
CGF
INPUT FILE HANDLER
FIND ASSERTION
SC
FKWORD
VM
VM
ASRT
VM
CGF
SOURCE FILE
FIND KEYWORD
CHECK ASSERTION FORM

FIGURE 2.1

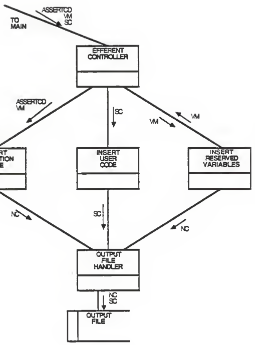

ASSERTCD
VM
SC
TO MAIN
EFFERENT CONTROLLER
ASSERTCD
VM
SC
VM
VM
INSERT ASSERTION CODE
INSERT USER CODE
INSERT RESERVED VARIABLES
NC
SC
NC
OUTPUT FILE HANDLER
NC
SC
OUTPUT FILE

FIGURE 2.1

Assertion Code Creator and Inserter

Hierarchy Specifications

```
Procedure:              Main
    calls:              Afferent_controller
    calls:              Transform_controller
    calls:              Efferent_controller
    input_global:       $Validity_messages$
    input_parameter:    $Correct_general_forms$
    input_parameter:    $Variable_declarations$
    input_parameter:    $Source_codes$
    input_parameter:    $Assertion_codes$
    output_global:      $Validity_messages$
    output_parameter:   $Correct_general_forms$
    output_parameter:   $Variable_declarations$
    output_parameter:   $Source_codes$
    output_parameter:   $Assertion_codes$


Procedure:              Afferent_controller
    calls:              Get_filename
    calls:              Find_comments_and_variable_
                        declarations
    input_global:       $Validity_messages$
    input_parameter:    $Valid_fns$
    input_parameter:    $Source_codes$
    input_parameter:    $Variable_declarations$
    input_parameter:    $Correct_general_forms$
    output_global:      $Validity_messages$
    output_parameter:   $Valid_fns$
    output_parameter:   $Correct_general_forms$
    output_parameter:   $Variable_declarations$
    output_parameter:   $Source_codes$


Procedure:              Get_filename
    ext_input:          $Filenames$
    ext_output:         $File_errors$
    output_parameter:   $Valid_fns$
```

Figure 2.2
Unknown example ERL

```
Procedure:                      Find_comments_and_variable_
                                declarations
     calls:                     Input_file_handler
     calls:                     Find_assertion
     input_global:              $Validity_messages$
     input_parameter:           $Valid_fns$
     input_parameter:           $Correct_general_forms$
     input_parameter:           $Source_codes$
     output_global:             $Validity_messages$
     output_parameter:          $Source_codes$
     output_parameter:          $Variable_declarations$
     output_parameter:          $Correct_general_forms$
     output_parameter:          $Valid_fns$
     output_parameter:          $Comments$


Procedure:                      Input_file_handler
     input_parameter:           $Valid_fns$
     ext_input:                 $Source_codes$
     output_parameter:          $Source_codes$


Procedure:                      Find_assertion
     calls:                     Find_keyword
     calls:                     Check_assertion_form
     input_global:              $Validity_messages$
     input_parameter:           $Comments$
     input_parameter:           $Correct_general_forms$
     output_global:             $Validity_messages$
     output_parameter:          $Words$
     output_parameter:          $Assertions$
     output_parameter:          $Correct_general_forms$


Procedure:                      Find_keyword
     input_parameter:           $Pkwords$
     output_global              $Validity_messages$


Procedure:                      Check_assertion_form
     input_global:              $Validity_messages$
     input_parameter:           $Assertions$
     output_global:             $Validity_messages$
     output_parameter:          $Correct_general_forms$
```

Figure 2.2
Unknown Example ERL

```
Procedure:                    Efferent_controller
     calls:                   Insert_assertion_code
     calls:                   Insert_user_code
     calls:                   Insert_reserved_variables
     input_global:            $Validity_messages
     input_parameter:         $Assertion_codes
     input_parameter:         $Source_codes
     output_parameter:        $Assertion_codes
     output_parameter:        $Source_codes
     output_global:           $Validity_messages

Procedure:                    Insert_user_code
     calls:                   Output_file_handler
     input_parameter:         $Source_codes
     output_parameter:        $Source_codes

Procedure:                    Insert_assertion_code
     calls:                   Output_file_handler
     input_global:            $Validity_messages
     input_parameter:         $Assertion_codes
     output_parameter:        $New_codes

Procedure:                    Insert_reserved_variables
     calls:                   Output_file_handler
     input_global:            $Validity_messages
     output_parameter:        $New_codes
     output_global:           $Validity_messages

Procedure:                    Output_file_handler
     input_parameter:         $New_codes
     input_parameter:         $Source_codes
     ext_output:              $New_codes
     ext_output:              $Source_codes
```

Figure 2.2
Unknown Example ERL

```
Procedure:                      Transform_controller
     calls:                     Append_to_variable_table
     calls:                     Create_assertion_code
     input_global:              $Validity_messages
     input_parameter:           $Variable_declarations
     input_parameter:           $Correct_general_forms
     input_parameter:           $Var_tables
     input_parameter:           $Assertion_codes
     output_global:             $Validity_messages
     output_parameter:          $Variable_declarations
     output_parameter:          $Correct_general_forms
     output_parameter:          $Var_tables
     output_parameter:          $Assertion_codes

Procedure:                      Append_to_variable_table
     input_parameter:           $Variable_declarations
     output_parameter           $Var_tables

Procedure:                      Check_assertion_code
     calls:                     Check_assertion_syntax
     input_global:              $Validity_messages
     input_parameter:           $Correct_general_forms
     input_parameter:           $Var_tables
     input_parameter:           $Correct_assertions
     output_global:             $Validity_messages
     output_parameter           $Correct_general_forms
     output_parameter           $Var_tables
     output_parameter           $Assertion_codes

Procedure:                      Check_assertion_syntax
     input_global:              $Validity_messages
     input_parameter:           $Correct_general_forms
     input_parameter:           $Var_tables
     output_global:             $Validity_messages
     output_parameter           $Correct_assertions

Input:                          $File_names
    media:                      Keyboard
    structure:                  $Strings
    c_dec:                      char file_name[15]
```

Figure 2.2
Unknown Example ERL

```
Input Input_output              $Source_code$
    media:                      secondary storage
    structure:                  $Code_files$
    structure:                  $Assertion_strings$
    c_dec:                      char source_code[80]

Output Input_output:            $New_code$
    media:                      secondary_storage
    structure:                  $Code_files$
    c_dec:                      char new_code[80]

Input_output:                   $Error_msg$
    structure:                  $Strings$
    c_dec:                      char *error_msg =
                                      "NOT ABLE TO OPEN FILE"

Input_output:                   $Valid_fn$
    structure:                  $Strings$
    c_dec:                      char valid_fn[15]

Input_output:                   $Comments$
    structure:                  $Strings$
    c_dec:                      char comments[80]

Input_output:                   $Comment_remainder$
    structure:                  $Strings$
    c_dec:                      char comment_remainder[80]

Input_output:                   $Assertion$
    structure:                  $Strings$
    c_dec:                      char assertion[80]

Input_output:                   $Var_table$
    structure                   {$Var_declaration$}
    c_dec:                      struct {var_dec[80]} table_entry
                                      table_entry var_table[30]
```

Figure 2.2
Unknown example ERL

```
Inout_output:                  $Correct_assertion$
    structure:                 $Assertion_parts$
    c_dec:                     struct {cnar keyword[15]
                                       int lower_bound
                                       int upper_bound
                                       cnar logic_comparison[2]
                                       int comparison_number
                                       cnar boolean_expression[30]
                                     } assertion_type
                               assertion_type correct_assertion


Input_output:                  $Correct_general_form$
    structure:                 $String$
    c_dec:                     cnar correct_general_form[80]


Input_output:                  $Variable_declaration$
    structure:                 $Var_declaration$
    c_dec:                     struct {cnar variable_type[80]
                                       char variable_name[80]
                                     } dec_type
                               dec_type variable_declaration


Input_output:                  $Assertion_code$
    structure:                 $Code_string$
    c_dec:                     typedef cnar string[80I
                                   string assertion_code[25]


Global:                        $Validity_message$
    structure:                 $Validity_string$
    c_dec:                     int validity_message


Type:                          $Var_declaration$
    structure:                 $variable_type$,{$variable_name$}


Type:                          $Variable_type$
    structure:                 string


Type:                          $Variable_name$
    structure:                 string
```

Figure 2.2
Unknown Example ERL

```
Type:                       $String$
    structure:              string


Type:                       $Assertion_string$
    structure:              $Boolean_string$
    structure:              $Some_string$
    structure:              $For_string$
    structure:              $Numberof_string$
    structure:              $Control_string$


Type:                       $Assertion_parts$
    structure:              $Keyword$
             :              $Lower_bound$
             :              $Upper_bound$
             :              $Logic_comparison$
             :              $Comparison_number$
             :              $Boolean_expression$


Type:                       $Boolean_string$
    structure:              '(BOOLEAN'
             :              $Boolean_expression$
             :              ')'


Type:                       $Some_string$
    structure:              '(SOME:'
             :              $range$
             :              $Boolean_expression$
             :              ')'


Type:                       $For_string$
    structure:              '(FORALL:'
             :              $range$
             :              $Boolean_expression$
             :              ')'
```

Figure 2.2
Unknown Example ERL

```
Type:                           $Numberof_strings
    structure:                  '(NUMBEROF:'
             :                  $ranges$
             :                  $Logic_comparisons$,
                                $Integer_numbers$
             :                  ':'
             :                  $Boolean_expressions$
             :                  ')'


Type:                           $Control_strings$
    structure:                  '(ASSERTIONOFF'
             :                  $Boolean$
             :                  ')'


Type:                           $Boolean$
    structure:                  '=true'
    structure:                  '=false'


Type:                           $Boolean_expressions$
    structure:                  string


Type:                           $Ranges$
    structure:                  $Lower_bounds$
             :                  '..'
             :                  $Upper_bounds$
             :                  ':'


Type:                           $Logic_comparisons$
    structure:                  '<'
    structure:                  '>'
    structure:                  '<='
    structure:                  '>='
    structure:                  '=='
    structure:                  '!='


Type:                           $Lower_bounds$
    structure:                  $Integer_numbers$


Type:                           $Upper_bounds$
    structure:                  $Integer_numbers$
```

Figure 2.2
Unknown Example ERL

```
Type:                           $Keywords$
    structure:                  'SOME'
    structure:                  'FORALL'
    structure:                  'BOOLEAN'
    structure:                  'NUMBEROF'
    structure:                  'ASSERTIONOFF'
    c_dec:                      char keyword[15]


Type:                           $Comparison_numbers$
    structure:                  $Integer_numbers$


Type:                           $Integer_numbers$
    structure:                  $Integer_constant$
    structure:                  $Integer_variables$


Type:                           $integer_constant$
    structure:                  integer


Type:                           $Integer_variables$
    structure:                  $Variable_names$


Type:                           $Code_files$
    structure:                  file of $Strings$


Type:                           $Code_strings$
    structure:                  {$Comments$}
             :                  [$codes$]


Type:                           $Validity_strings$
    structure:                  'SOME_assertion'
    structure:                  'FORALL_assertion'
    structure:                  'BOOLEAN_assertion'
    structure:                  'NUMBEROF_assertion'
    structure:                  'ASSERTIONOFF_assertion'
    structure:                  'comment_only'
    structure:                  'invalid_assertion'
    structure:                  'declaring_variable'


Type:                           $codes$
    structure:                  {$string$}
```

Figure 2.2
Unknown Example ERL

module.

The facts that allow me to conclude that this is a coordinate module are:

1. The top level of the hierarchy is almost always (.95) a coordinate module.
2. The word "main" indicates (.9) coordination.
3. All the outputs are the same as the inputs.
4. More than one module is called.
5. There are no external inputs.
6. There are no external outputs.

The certainty factors listed with the first two facts are first approximations, as are all the certainty factors in this paper. After the expert system is implemented these probabilities will be adjusted to maximize agreement between the expert system evaluation and the judgment of the human evaluators of designs.

2.3.2 Afferent Controller module

The analysis of design includes the evaluation of the collaboration between the modules; thus, a breadth first examination of the hierarchy is the appropriate method to use.

The module Afferent_controller also has all inputs passed through as outputs. The root word "control" tells me that this is a coordinate module, also supported by the facts that it is in the second level of the hierarchy and that it calls more than one other module. Obviously the students were recently exposed to the concepts of afferent and efferent flow when they named their modules. Although I wouldn't expect these names, I could anticipate names denoting I/O like input, read, get, output, print and write. Programs generally have stages of input, process and output. Designs also are developed in this fashion and generally proceed left to right.

The facts used to determine the nature of Afferent_controller are:

1. The second level of the hierarchy is likely (.8) to include coordinate modules.
2. The word "control" embedded in the module name indicates (.95) that this is a coordinate module.
3. All the outputs are the same as the inputs.
4. More than one module is called.
5. There are no external inputs.
6. There are no external outputs.

Therefore this module is a coordinate module. Because this module is the first module in the calls slot, I can say:

7. This is the left most module in the second level

of the hierarchy.

8. The module on the left of the second level of the hierarchy is generally (.8) an afferent module.
9. The module returns more data into the system than it receives.

Afferent_controller is part of the afferent portion of the design.

2.3.3 Transform Controller module

The next module to analyze is the Transform_controller. Note that I select the search based on the order of the calls as they appear in the Main module and not on the order of appearance in the ERL specification. In this example the order of Transform_controller and Efferent_controller is switched in the document. Disregarding the information obtained from the names of the modules, I would question the succession of the modules because the order in which they are listed differs from the sequence listed in the calling module. By choosing the next module by order of appearance in the calling module I expect to be correct more often than if I chose by the order of appearance in the document. The justification of this heuristic is that the information is more local in the module than in the document. In automating the analysis of design the switching of the order would prompt the expert system to question which is the more proper order. Selection by the sequence in the calling module is the default sequencing.

My conclusion of the purpose of the Transform_controller is that it is a coordinate module. The following facts are used to support this conclusion.

1. The second level of the hierarchy is likely (.8) to consist of coordinate modules.
2. The word "control" embedded in the module name indicates (.95) that this is a coordinate module.
3. All the outputs are the same as the inputs.
4. More than one module is called.
5. There are no external inputs.
6. There are no external outputs.

Additionally this module is in the middle 50 percent of the second level of the hierarchy, determined by its position in the calls slot of the Main module in the ERL. Therefore it is likely to be part of the transform section of the design. Also, it receives more information than it returns to the program.

2.3.4 Efferent Controller module

The examination of the Efferent_controller module will follow the same lines as the previous two modules. I've determined the Efferent_controller to be a coordinate module of the efferent section because:

1. The second level of the hierarchy is likely (.8) to consist of coordinate modules.
2. The word "control" embedded in the module name indicates (.95) that this is a coordinate module.
3. All the outputs are the same as the inputs.
4. More than one module is called.
5. There are no external inputs.
6. There are no external outputs.
7. The module on the right of the hierarchy is generally (.8) an efferent module.
8. No data are returned to the parent module.

Now that all the activities across the second level have been examined, it is wise to look across the breadth of this level to evaluate how well the modules link together. All the modules have been judged as coordinate modules, and I have even gone so far as to place afferent, transform and efferent labels on each of them. This is consistent with what the model expects for a design, partially because the design forced some labels on the module. Further support is needed to establish the correctness of these decisions.

2.3.5 Get Filename module

Proceeding to the next level of the hierarchy I take the first module called from the Afferent_controller; the Get_filename activity. To a human, the intent of Get_filename is obviously to obtain the name of an input file. What are the facts that allow one to determine its function?

1. There is an external input.
2. There is an external output that has "error" as part of its name.
3. The word "get" occurs in the name of the module.
4. A correspondence of words in the input of the module and the name of the module makes it likely (.7) that input is a major function of this module.
5. The structure of the external input is the same as the output from the module.
6. The parent module is a coordinate module for the afferent section.
7. The module passes more information back to its parent than it receives.
8. No other modules are called from this module.

Therefore, I can say that Get_filename is an afferent module whose purpose is to get primary input from the keyboard, the medium of the external input. Since the input medium is keyboard, an expectation is developed that a prompt should be generated. A human evaluator of the design would make a note about the keyboard prompt, and an expert system should do the same.

Another point needs to be clarified about this module: What conditions are going to trigger the error message? This is why I will add the intrinsic knowledge field to the ERL specification so a slot is available to contain this information, thereby making the ERL a more complete document. The intrinsic knowledge slot would contain information like "if filename cannot be opened then print 'file_error'". The format is that of a test and the error message is the result of the test. This will satisfy the expectation that the conditions triggering the error message are stated in the ERL. At this time, just a check will be made to see if such a statement is present, deferring the attempt to obtain any further meaning from the condition statement for future work.

2.3.6 Find Comment and Variable Declarations module

The next module to examine is Find_comment_and_variable_declarations. The items I can determine from this module are:

1. All inputs are also outputs, but there are two

additional outputs.

2. The name of a new output, Comment, appears in the name of the module.
3. The name of a new output, Variable_declaration appears in the name of the module.
4. There are no external inputs.
5. There are no external outputs.
6. The module passes more information to its superordinate than it receives.

Therefore, the classification of this module is a choice between transform, because of the changes of input to a different output, and afferent, because more data are passed into the system. By drawing on the environment, the system will be influenced by the identification of the sibling and parent modules and declare this module afferent. However, the confidence of this choice will be low (around fifty percent).

Inspection of this module reveals a problem. $Comment$ is not declared in the data type, but $Comments$, with the 's' is. I did not notice this error until the nth time through the ERL specification: n greater than four. This demonstrates one of the needs for automated analysis of design.

The above error is easily overlooked and automatically corrected by a human. It also should not be a stumbling block for an expert system. The omission or addition of an extra letter(s) to give the plural is such a common error that an automated system should be able to recover from it, print a warning message and not grind to a halt.

$Comments$, besides being an Input_output, is traced through the data structure to be a part of $Code_string$, and $Code_string$ is the structure of $Assertion_code$. $Assertion_code$ is used as a variable in the modules Main, Transform_controller, Efferent_controller, Insert_assertion_code and Check_assertion_code. It is not connected to the afferent section or Find_comment_and_variable_declaration. The above trail leaves $Comments$ in limbo, and the matter needs to be brought to the attention of the user.

Following the paths of $Variable_declaration$ does not produce any clues about the origin of $Comments$ either. The trail ends at a string with no revealing non-terminals. The object of the search through the data structure is to find if either $Comments$ or $Variable_declaration$ was part of the structure of one of the inputs to the module.

Because these two modules are the only activities called from the Afferent_controller, a review of the compatibility between the modules is in order. Get_filename is resolved to be an afferent module, as is Find_comment_and_variable_declarations. Get_filename can also pass back the information that it is an afferent module. Although this is circular reinforcement of the fact that the parent was an afferent section module, this information should be exchanged so that the certainty factor of the beliefs can be adjusted accordingly. The certainty factor should not be adjusted by Find_comment_and_variable_declarations response that it is an afferent module because the certainity factor is less than already established for Afferent_controller.

2.3.7 Append to Variable Table module

Among the children of the Transform_controller, the comparison of the inputs and outputs of Append_to_variable_table is very interesting. The other information obtained from the ERL entry is the word "append" in the module name. Var_table, the output, is an array of Variable_declarations, the input. Since an array is a way of implementing a collection of elements, and "append" is associated with adding information to a collection, the pieces fit together. Establishing the intermediary of the Variable_declaration as the element of the collection to be inserted, I can justify that an insertion operation is the most likely intent of the module. The difficulty is that the array, Var_table, is not passed into this module.

I can think of two explanations for what must be the intent of this module. First is that the ERL is correct and a number of Variable_declarations are passed to the module and tbe Var_table is constructed and passed back only wben completed. Tbis would make this module fit nicely under the transform section and would also explain why there is no array index passed into the procedure.

The problem with that assessment is: Where do the numerous Variable_declarations come from? Since the Var_table is an array of Variable_declarations, it appears that its purpose must be the collection of Variable_declarations. Therefore, another explanation of what the module should be is that Var_table was omitted as an input. With that correction, the module is seen as building up a data structure, and therefore must be an afferent module. Missing is tbe array index as input, which is a common error, especially if the array is absent as input.

The facts of Append_to_variable_table are:

1. Tbe input is an element of the output structure.
2. The word "append" appears in the activity name.
3. The information provided does not fit the expectations of an append module.

This module should be flagged as undetermined, and the user should be questioned to determine if the Var_table was overlooked as an input or if the first alternative was actually correct. If the ERL entry is correct, the intrinsic knowledge field can be used as a flag and set to "confirmed" to prevent the assessment from reproducing the problem on subsequent iterations.

2.3.8 Create Assertion Code module

Create_assertion_code is found to have one output that is different than the input. The others are passed through to the one module that it calls. The structures of the input and output are different, but there is a similar word, "assertion" in Correct_assertion and Assertion_code, signifying some type of correspondence. Whether the other inputs are used in this module to generate the Assertion_code cannot be determined.

I evaluate Create_assertion_code to be a transform module because:

1. There are no external inputs.
2. There are no external outputs.
3. At least one output differs from the inputs.
4. The word "create" occurs in the module name.
5. The word "assertion" occurs in the new output and in the module name.

6. The word "assertion" occurs in the one input that is not an output.
7. The parent module is a transform module.

Among the children of Transform_controller, there is Append_to_variable_table which has been labeled as undetermined and Create_assertion_code which has been recognized as a transform module. Create_assertion_code will support the deduction of the parent module as a transform activity. The evidence to go back and label the Append_to_variable_table is there, but a strong counter argument exists with the inputs and outputs that the purpose of the module is to build a data structure. Generating a data structure is considered to be an afferent activity. The decision is best put off until the questions are resolved.

2.3.9 Insert Assertion Code module

The first child of the Efferent controller is Insert_assertion_code. The inputs are Validity_message and Assertion_code, and the output is New_code. Because New_code is labeled as output from the program and in view of the fact that the parent module is believed to be efferent, this module is also labeled as efferent. The word "insert" in the procedure name generates uncertainty about the purpose of this module. New_code is traced through Code_file to have a structure of file of string, and file is a type of collection. The type declaration of New_code is an external output, but insertion is not an operation one expects to find in the efferent section of a system. The concept of the module is easy to determine. After "insert" in the name is the name of one of the inputs to the activity. The variable type of Assertion_code is string, which can be an element of the collection called New_code.

The facts present in this procedure are:

1. The output eventually becomes an external output.
2. New_code can be a collection of Assertion_code.
3. Insert is an allowable operation on a collection of items.
4. The parent module is a controller for the efferent section.
5. This module is the first introduction of the variable New_code.

Using facts 1, 4, and 5, I conclude the module is an efferent module. Facts 2 and 3 indicate the purpose is to add elements into a collection.

2.3.10 Insert User Code module

The Insert_user_code module presents an interesting set of facts for analysis. The inputs and outputs are the same, a major indication that the activity is used as a coordinate module, yet there is only one process called by the procedure. When there is only one child a controlling process is not needed, notably if the parent activity is a coordinate module. Designs generally do not duplicate effort. The analysis of Insert_user_code determines that it is useless because:

1. All the outputs are the same as the inputs.
2. Although Source_code can be an external input, it has been passed to this module, therefore there are no external inputs.
3. There are no external outputs.
4. There is only one called module.

2.3.11 Insert Reserved Variable module

I have difficulties judging Insert_reserved_variable. It has New_code as output, which eventually becomes external output, and it calls the same module as its sibling processes, so I am confident in labeling it as an efferent module like the others in this section. The problem is how New_code is generated from the Validity_message. The Validity_message is passed into numerous modules, making its use perplexing. The structure of this data is a string constant, indicating it is used as a flag. Validity_message is passed into Insert_reserved_variable and it is also passed back to the parent. Tbis leads me to think tbat something is happening to the Validity_message inside the module, and that New_code is being generated. Recalling that the sibling module Insert_assertion_code had Validity_message as input and New_code as output, I reason tbat there is some difference in the output of the two activities. I would question whether a similar additional input is needed for Insert_reserved_variable. There are no variables used with the name Reserved_variables to make a tight coupling between the two activities. The word "reserved" does not appear in any of the variable names. However, "variable" appears in the data declaration of Variable_declaration. A link of Variable_declaration to Var_table has been established from Append_to_Variable_Table so it also may be the possible missing input. The user would be queried whether information is missing from this module, either as input, suggesting Variable_declaration and Var_table, or as intrinsic knowledge such as a mapping from one state of Validity_message to another.

The facts obtained from this module are:

1. The output eventually becomes an external output.
2. New_code is a collection of elements of type string.

3. Validate_message is of type string and therefore could be added to New_code.
4. Insert is an allowable operation on a collection of elements.
5. The parent module is a part of the efferent section.
6. A sibling module is determined to be efferent.
7. A sibling module is determined to insert elements into New_code.

Therefore, Insert_reserved_variables is an efferent module and it adds elements into a collection. A note is made to question what information triggers a change in Validity_ message and whether Variable_declaration or Var_table might be an expected input.

A considerable part of the dissection of Insert_reserved_ variables involves looking back to one of its sibling modules. This should point to the need to examine the breadth of all the children of a module at each level to determine the cooperation among the siblings. Had the order of the sibling modules been reversed, the information that could be provided by going back over the entire group would have been available at this stage. Of course, since considerable knowledge was transferred from the first child to the third child, less reliance should be made on the information passed back. Two modules were evaluated to be efferent, and this detail can be returned to the parent to increase the confidence that it is a part of the efferent section. The declaration of the other child as useless does not weaken that conclusion.

Also a piece of information can be passed down the hierarchy. Since the last three modules call the same module, a label needs to be passed down that it potentially is a utility module. Utility modules are harder to write, because they must interface with more than one parent. When examining Output_file_handler the system will have to be more careful.

2.3.12 Input File Handler module

In the next level of the hierarchy, the Input_file_handler is determined to be an afferent module because:

1. There is an external input.
2. The external input is passed directly as output to the calling module.
3. The word "input" occurs in the name.
4. The word "file" in the name suggests (.7) that the use is I/O.
5. The ancestors are believed to be afferent modules.

2.3.13 Find Assertion module

Scrutinizing the Find_assertion module I note that the inputs differ from the outputs. The output appears to be generated from the inputs, and some type of string-to-string conversion takes place. The word "find" in the name suggests some type of search through the string.

An error occurs in this module which halts complete analysis until resolved. The data type of output $Word$ is not defined in the ERL. Additionally it is output from Find_assertion but neither the parent nor the two called modules have $Word$ as an input. Inspection continues and reveals that something happens to $Comments$ and that $Assertion$ and $Word$ are generated in the module. The two data types known do not have any intersecting non-terminals in the data structure portion of the ERL and eventually are resolved to be strings.

With the facts obtained:

1. The outputs are different from the inputs.
2. "Find" implies a search (.7), but since no tables, trees, lists are input, a parse of a string must be the search.
3. A word in the name matches a word in the output.

I judge this module to be a transform module to conduct a parse of $Comments$ producing $Assertion$ and $Word$, whatever $Word$ is used for.

Input_file_handler and Find_assertion send conflicting data back to their common parent Find_comment_and_variable_declaration. One is definitely afferent and the other is marked transform, slightly raising the confidence that the parent is afferent.

2.3.14 Check Assertion Syntax module

Check_assertion_syntax is the only called process of Create_assertion_code. I've evaluated this procedure to be a transform module because:

1. The structure of the inputs contains parts of all of the output.
2. A word in the name matches a word in one of the outputs.
3. Its ancestors are transform modules.

2.3.15 Output File Handler module

Output_file_handler is a module that is called from several sources, in other words a fan-in greater than one, and as mentioned earlier will have a entry that it is a utility module for that reason. Output and file as words in the name give clues about the purpose of this activity. The inputs are the same as the outputs and because the outputs are external, this must be an efferent module. Once the purpose is established to output to a file an expectation is generated that a filename is needed. This module does not have one provided, and the system has not encountered an efferent procedure that has a filename. Contrast the above to the afferent section that had a module to obtain a file name.

The facts extracted from this module are:

1. It is an utility module.
2. There are external outputs.
3. The inputs go directly to outputs.
4. The word "output" occurs in the name.
5. The word "file" occurs in the name.

Output_file_handler is an efferent module to write to a file, yet missing a filename, either default, inputted, or identical to the input filename. No matter what the case the filename should be specified on the ERL document. This is another example of the use for the intrinsic knowledge slot. If the project outputs to a standard file, then the intrinsic knowledge slot would look like "$filename$ = 'out.file'".

2.3.16 Find Keyword module

Find_keyword at the next level of the hierarchy in the afferent section is the originator of the $Validity_message$. The input $Pkword$ is not defined and there rests a problem with determining this module's purpose. The only interpretation available is undetermined. A message describing the missing data about $Pkword$ should be printed. A weak connection can be made to the parent, Find_assertion. That module has an output, $Word$, which is not defined or supplied to any module. Humans can recognize a link between $Word$ and $Pkword$, but the pattern recognition scheme necessary for automated analysis is more complex than that for detection of a missing "s" in the case of $Comment$ described earlier.

2.3.17 Check Assertion Form module

The last module in the hierarchy to be evaluated is Check_assertion_form. The inputs of $Validity_message$ and $Assertion$ apparently map to give the output $Correct_general_form$. This is supported by the fact that $Correct_general_form$ and $Assertion$ are strings and $Validity_message$ is one of a certain type of string. Because $Validity_message$ is passed back, the possibility exists that it may be changed in this procedure.

The facts are:

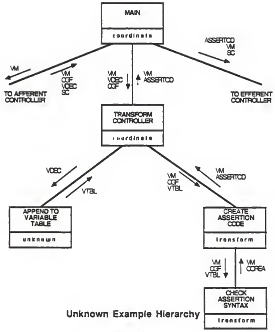
1. At least one output is different from the inputs.
2. There are no external inputs.
3. There are no external outputs.
4. Correct_general_form can be created from the inputs.
5. A word of the name matches a word in the different output.
6. The parent is a transform module.

Therefore, this is a transform module that does a mapping to obtain Correct_general_form. The analysis reveals the structure depicted in Figure 2.3.

2.3.18 Review

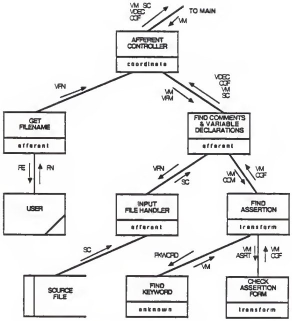
In the whole system, a data type, $Comment_remainder$, is defined, but never used, another point to bring to the attention of the designer. No judgment can be offered whether the omitted data item is extraneous, was hidden by remaining contained in a complex module, or was overlooked in two or more modules.

The overall shape of the design has one first level module, three second level, seven third level modules, four modules on the fourth level and two modules at the fifth lev-



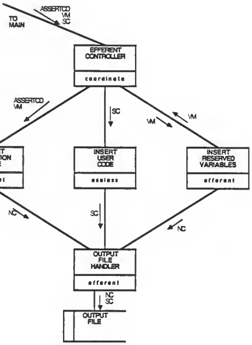
Unknown Example Hierarchy

FIGURE 2.3



VM SC
VDEC
CGF
TO MAIN
VM
AFFERENT CONTROLLER
coordinate
VFN
VDEC
CGF
VM
SC
VM
VFM
GET FILENAME
afferent
FIND COMMENTS & VARIABLE DECLARATIONS
afferent
FE
FN
VFN
SC
VM
COM
VM
CGF
USER
INPUT FILE HANDLER
afferent
FIND ASSERTION
transform
SC
PKWORD
VM
VM
ASRT
VM
CGF
SOURCE FILE
FIND KEYWORD
unknown
CHECK ASSERTION FORM
transform

FIGURE 2.3



TO MAIN
ASSERTCD
VM
SC
EFFERENT CONTROLLER
coordinate
ASSERTCD
VM
SC
VM
VM
INSERT ASSERTION CODE
efferent
INSERT USER CODE
useless
INSERT RESERVED VARIABLES
efferent
NC
SC
NC
OUTPUT FILE HANDLER
efferent
NC
SC
OUTPUT FILE

FIGURE 2.3

el. This has a bulge in the middle and more tapering at the lower end than expected, suggesting that detail was omitted from the design. The depth of the hierarchy under the afferent section is five and the depth under the transform and efferent sections is four. The evidence is slightly indicative of an input intensive system. A simple count lends support to this expectation. There are six modules under the afferent controller, three under the transform controller and four under the efferent controller. This evidence would support the hypothesis that the process is input intensive. A count of the labels analysis attached to the modules points to a different interpretation. Four modules have an afferent label, five have a transform label, four are marked as efferent and three modules are unknown or useless. The balance observed points to a normal transform system.

2.4 Remarks

Knowledge of the source of inputs and destination of outputs can be a significant aid in the evaluation of the purposes of the modules. The first pass an expert system should make of a design is to traverse the hierarchy and establish the paths of the data by matching subordinate inputs to superordinate outputs and vice versa. The notation showing which data items were externally inputted and outputted was helpful. After this initial pass, the system should generate a hierarchy graph with the data items labeled, so a designer may compare what the ERL states against what he or she intended it to say. In this example, the Validity_message would not be detected as being passed down in the afferent section, but only passed up, originating in Find_keyword and used in Check_assertion_form. Also a graph of all the different data structures should be made. The graph will help determine how certain data items can be created and how others are used. In this example, if the connection between Source_code and Comments would have been established, the task of analyzing Find_comments_and_variable_declarations would have been easier.

The results of this evaluation were not given to the team that created the design. A test of this thought experiment will be to compare the difficulties discovered in the analysis with the hierarchy of the project determined from the code and not from the documentation and to see if changes occurred where problems were spotted.

Chapter 3

Examination of the Rules

The example of the previous chapter was selected after a preliminary review of several designs. A scan of these designs shows that some facts produce consistent results while others indicate a trend, and yet others are only good for invalidating a conclusion.

In this chapter the Yourdon - Constantine model will be discussed in terms of what information it provides to assist analysis. Additional information provided from studies of the previous authors is also discussed. The results of an experiment conducted by Adelson and Soloway with expert and novice designers will be examined to see how the expert design knowledge can be incorporated into the automated system.

3.1 Yourdon Constantine Model Shape

The shape of the example in chapter 2 does not fit the expected shape of the Yourdon Constantine model. Two explanations contribute to this behavior. First, the students are dealing with relatively small software projects, therefore, shallow but broad hierarchy diagrams are the result. A shallow hierarchy does not have a well established shape. The second reason the shape of the student's designs fails to fit the model's predicted shape is that students are not likely to develop the detail expected of real-world designs. I am inclined to believe that each shop needs to inspect its data for the shapes its designs have before establishing the parameters expected on the shape.

I believe that the rules that judge a design on the basis of its shape are not as reliable as others.

3.2 Module Names

Many of the facts used to determine the function of an activity are based on words extracted from the name of the module. Several naming conventions are in general use in software design, including the use of a consistent set of names by design teams. These names can be programmed to be associated with certain functions or types of functions. Names are not meant to indicate specific processes but indicate generic processes in general terms. That is also how this expert system will analyze a software design. Table 3.1 from Yourdon - Constantine shows names that are commonly used along with the module type that is associated with the name. The guidance provided by module names will be discussed further in the next chapter.

3.3 Rule Confidence

Module Names

Afferent processes with external sources of data

GET	ACCEPT (usually asynchronous)
OBTAIN	FIND
INPUT	LOAD

Afferent processes with internal sources of data

SETUP	FORM
DEVELOP	CREATE
GENERATE	

Transform processes

ANALYZE	COMPUTE
TRANSFORM	CALCULATE
CONVERT	PERFORM
DO	PROCESS

Specific verbs like SORT, VALIDATE, etc.
Function-oriented nouns like SQUAREROOT, INVERSION

Efferent processes with external targets

PUT	OUTPUT
PRODUCE	STORE
SAVE	WRITE
DELIVER	PRINT

Table 3.1

Throughout the examination of designs for rules, attention was paid to the universality of the generated rule. The rule must be conclusive for the module and must not contradict other rules already proposed. Additionally, the rules must be applicable to other designs.

For example, in trying to develop a rule for determining if a module was a coordinate module, one item I looked at was the number of called activities and the number of mode transitions allowed in the module. While examining other designs, I discovered that many of the student designs only had a fan-out of two from many controlling modules, whereas the first design studied had typical fan-outs of four. Thus, the number of called activities is not important in determining if a module is coordinating. One fact that is consistent across all designs was that if the inputs were directly passed as outputs I had also determined the module to be a coordinating activity.

One must be careful when attempting to determine the category of a module based upon the external I/O. For example, the Get_filename activity has both external input and external output, but its purpose can not be both afferent and efferent. The task is to determine the intent of the activity using information extracted from the inputs and outputs.

An external output that is an error message cannot be used to classify an activity since any activity may generate a message communicating that an error condition was detected. External outputs that are prompts belong to the afferent section because they are used to request more information for the system.

The indication that the external input or output is part of an afferent or efferent module respectively would be if its medium is secondary storage. Evidence of considerable amounts of input to or output from outside sources, ie. a file or array data type, also implies that the module belongs to the afferent or efferent portion of the system, respectively.

A good measure of a module being afferent or efferent is a comparison of the count of items the activity passes to its superordinate to the number of items it receives from its parent. Afferent modules will pass more information back to the parent, and efferent modules receive more data than they return. These facts are part of the definition of afferent and efferent. I noticed these facts give more reliable clues near the top of the hierarchy than they do at the bottom.

3.4 Matching Inputs and Outputs

A significant amount of programming is associated with construction of data structures, extraction of particular information from a data structure, and validation of data. A comparison of the data types of the inputs and outputs of an activity and the discovery of a common data type, if any, is essential in determining the function of an activity.

In the example, the function of Append_to_Variable_Table was easy to determine. Since Variable_declaration, as input, is the type of the elements of the array Var_table, which is output, a deduction is confidently made that the activity builds an array data structure. The difficulty of fitting the activity into the design was the result of following up on the conclusion.

Examination of the data types of the inputs and outputs benefits analysis by revealing if a data structure is being erected or if information is extracted. The check of elements or fields of a data structure assists the examination of modules that manipulate arrays, lists, graphs, stacks, files, records, etc. There is a limited set of operations allowed on each data structure; if the possible functions of a module are limited to a subset of those operations, then the determination of what the module is attempting to do is simplified.

Once a certain operation on a data structure is recognized, expectations are developed that complementary operations also will be performed. The example in chapter 2 includes the construction of an array which is used in only one other module. One would expect to see the array enter a coordinating module from where the elements are passed to a called process where some calculation is executed. This is not the case in this example, supporting the idea that details of the design are missing.

The operations of numerous computer systems are manipulating data into aggregate types or mapping data structures to other data structures. The determination of what each module intends to accomplish can be made from a comparison of the data structures of inputs and outputs, and a search for a common data type.

3.5 Go Around Again

In analyzing designs from the ERL, the system can extract information from three entities. The data entity relates each data item used to its structure and source. The activity entity details the data items entering and exiting the module, a location in the hierarchy, some activity name, and/or some keyword. The mode entity, which was not used in the example of chapter 2 makes known the control flow of the system. These entities are related and reinforce each other. Once a set of facts is found present, a rule fires suggesting what the function of the module is. The function of the activity in turn triggers a rule to expect the presence of certain facts in the module. The circular path either generates confirmation of the module or raises questions to be resolved by the designer.

The mode entity reveals the control of the system. It lists all modes of the system, the allowed transitions between modes and the conditions that permit or trigger the transition. The ERL can establish the data flow, thus, an automated analysis can be performed to determine if the control flow matches the data flow. The data flow and control flow must be compatible because data flow is control flow with the added property that it specifies the transfer of data [Wa78]. In the Append_to_Variable_Table module of the previous example, two determinations were proposed as to what the purpose of the module is. If control information was also present, one argument may have been eliminated because it would violate the indicated mode transition. In the example, if the mode transition did not allow control to alternate between Find_comment_ and_variable_declaration and Append_to_variable_table, then the second proposed purpose of obtaining Variable_declaration and building the array, Var_table, would be eliminated, thereby clarifying the intent of the activity.

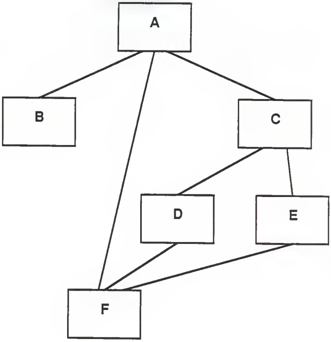
3.6 Breadth Examination of Designs and Abstraction Levels Maintenance

There was little use of complex data structures in the previous example, but some points need to be made about why breadth-first examination of the design is used.

A design hierarchy of a software project expressed in the textual form of the ERL may not hold the level structure that was intended by the designer. Figure 3.1 shows a structure that when expressed in the ERL evaluates activity F at the same level as activities B and C. A depth-first search would not catch the different level of activity F until it passed over F, C and D.

With the breadth-first scan of the system, after each section has been examined the levels of abstraction of the data types of the activities can be compared. If there is a significant difference, ie. if two modules are processing two-dimensional arrays and another sibling is working with a few integers, then the abstraction level of the hierarchy is skewed, and the activity dealing with the integers should be moved down the hierarchy to the level concerned with base data types.

Adjustment of the levels of the hierarchy by the abstrac-



Sample Hierarchy

Figure 3.1

tion level of the data type will aid the designer in several ways. First he or she can see if an activity is trying to work with data structures at too basic a level. In other words the system is fighting against the aggregation of data in the structure and is not making use of the operations designed to work on the data structure. Or the data structure was assembled before the pieces were ready. The second example of chapter 5 will demonstrate the matching of hierarchy levels with the data abstraction levels.

Breadth-first analysis is the correct method to use to ensure that the modules cooperate. The evaluation pertains not only to the function of a module, but also to its relation to the functions of its siblings. Activities called by the same parent must work in harmony if the system is to perform as the designer intends it to.

3.6.1 How Expert Designers Design

An experiment performed by Beth Adelson and Elliot Soloway [Ad85] illuminates how expert and novice designers develop systems. The authors discuss the differences between the styles of the two classes of designers. Of particular interest is the case where the designers were working in a domain in which they were familiar, but the object of the design was unfamiliar, the situation most designers find themselves in.

The expert designer begins at the high abstract level and progressively works to the more concrete. The experimenters noted that the designers worked on only one level at a time and incrementally expanded the details of the design. It was observed that the expert designers constructed a mental working model to simulate their design to check the consistency of all the possible inputs and unforeseen interactions. Additionally the interaction between the modules was simulated. During the simulation all the I/O elements have to be at the same level of detail. In Figure 3.1, the output from activity B cannot become input for activity C if the abstraction levels are not comparable. If the expert designer bad a concern about part of the design but it was at a different level of detail than was currently being worked on, he or she would make a note to be acted upon at the appropriate time.

One other activity was observed during the experiment that relates to automating analysis. The designers would detail the design to the point where they knew how the rest of the activities below would work. For example a designer would detail to an activity "sort", but since he or she already had a repertoire of sorting routines, the design process was considered finished along that branch. This indicates the expert system should have a library of pre-defined projects.

When the designers in the study were presented a task to design outside of the domain they normally work in, the experimenters noticed that the level of detail in the mental simulation was not as deep. Another observation was that the simulation of the design followed one input at a time, in isolation, without the interactions the designers were able to recognize when they were familiar with the design task.

As one would expect the designs of systems outside the expert designer's domain had bugs whereas, within their domain, the plans were essentially correct. One difference between novices and experts is in the simulation process -- the novices have representations whereas the experts develop models.

3.6.2 What to Include in the Expert System

By using breadth-first analysis the automated analysis will simulate the behavior of the expert designer by moving incrementally from the abstract to the concrete. It will examine the activities checking for unexpected interactions. However, at this point of development the knowledge in the automated analyzer is viewed as representational knowledge, like the knowledge of the novice. Once a domain and design team is established, relationships can be established between activities, the typical functions, and typical inputs so the machine can create a symbolic model and run a simulation as the expert.

Chapter 4

Data Structures

4.1 Introduction

A designer of a software system must be able to collect the pieces of information into manageable chunks. This leads to the development of data structures. Because data structures are designed for a computer scientist to assemble data in manageable pieces, it seems a natural way to have a machine analyze design schemes.

Data structures have also been proposed as a means for data hiding. Data hiding is an excellent idea and should be encouraged. The difficulty of this approach is that in the Yourdon - Constantine model of design, an afferent module builds the data structure, a transformation section does its manipulations, and an efferent part outputs the data. In this model and in others the knowledge about the data structure must be known throughout the system.

There are two ways of looking at data types; first one can look at the operations that can be performed on the data type, and second one can look at the characteristics of the data type. I have found that operations and characteristics are related. The terminology we use to refer to a data structure corresponds with what items we can manipulate through operations on a data structure. For example, a list and an array are very closely related; but because an array allows random access whereas a list must be traversed sequentially, we communicate about the index of an array. An index of a list would take further explanation to most programmers to describe exactly how you intend to use an index.

Data structures are defined so that each data structure has a unique set of characteristic values. A taxonomy was developed on the basis of the attributes that distinguish one data type from another. For example, the attribute that a real number is an approximation of a number and an integer is the exact value of a number is a distinction between the two numeric types. This analysis will make the addition of new data types to the taxonomy easier.

The operations on the data structures discussed in this paper are either primitives in the language, such as an assignment statement, or very common algorithms that have been amply discussed in the literature.

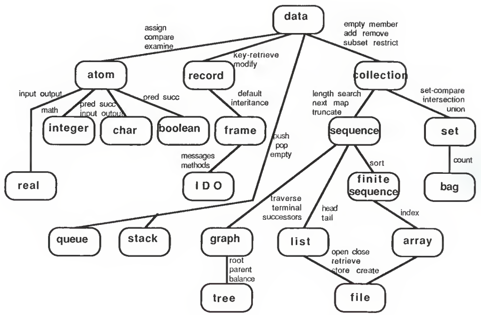
Under the assumption that programs are built from smaller pieces I have developed a taxonomy of data structures built upon the concept of primitive operations. Thus, the analysis of a design can proceed along the line of the data structure that enters a module, the type of structure that is output from a module, and the operations the module is intended to perform.

4.2 Operations

Each data type bas associated with it a set of operations that can be performed on or with that data type. As data objects become components of other data objects, they bring with them their associated operations, generally with the restriction tbat the component must be broken out of the structure before the operation can be performed. Tbe concept of adding two lists of integers togetber has two different implications, one result being a list that contains tbe catenation of both lists of integers, the second a list containing tbe sums of tbe corresponding integers. Thus, a major component of examining a design is to insure that the purpose of a module is to perform an operation that is appropriate for the types of inputs to the module.

Operations for the data types are well known. A brief description of the operations and associated data types that should be known by an expert system is included in Appendix A and summarized in Figure 4.1.

The most basic operation of a computer is the storage of a value to a variable and the examination of a variable to ob-



Hierarchy of data type by operations

Figure 4.1

tain the value stored there. Data types place attributes on the storage location to indicate the manner in which the bits are to be interpreted.

Another example to point out the need for consistency in the level of abstraction between the operations a module performs and the data structures that are associated with the module is stack multiplication. A module that has a stack of integers as input and a stack of integers as output, and an operation of multiply, does not make logical sense. Although many computer scientists would assume that the last two numbers would be popped off, multiplied and the product pushed back on the stack, the operation is not consistent with the data type stack. The need to show the actual inputs used by an activity is consistent with DeMarco' Functional Partitioning Rule [DM84] that a module should not have to decompose the tokens supplied to it.

Computer science is a rapidly expanding field, and hence new data types and operations are being created. Graphics has a set of data types -- line, arc, rectangle, etc -- and operations that include rotate, shrink, expand, fill, flash, etc. Similarly, data types will be developed in phonetics for speech understanding. Researchers in robotics also will develop data structures specific for their field. And the list will continue to grow. If a system is to perform a design analysis on the basis of data types and operations then a similar hierarchy of operations and related data structures will need to be appended to the expert system for the new application areas.

4.3 More Information about Operations

The expert system must have some pre-established expectations about operations on data types. A human has more information associated with the manipulations of data structures than just the name of the operation.

4.3.1 Operation Categorization

I have mentioned three major categories for modules in a design; afferent, transform and efferent. Certain operations are more likely to be found in one of these categories than in the others.

Modules with operations such as "retrieve" or "input" are almost always afferent modules. "Add" or "assign" modules are probably afferent modules, but may be transform. Modules to perform searches or to store data can be assumed not to be part of the afferent portion of the system.

Modules containing operations of mathematics, subsets, restricting, balancing, or modifying will probably be found in the transform section. Sections that work with semaphores, monitors, stacks and queues may also be expected to be transform modules.

Efferent modules are expected to perform store or output operations. Traversing and searching may be included in efferent operations, but those operations also may be found in the transform section of the design.

Not all operations lend themselves to this analysis, nevertheless the manipulations that are analyzed give the expert system the capability to handle more complex tasks.

4.3.2 Operation Pairs

Humans have even more information about data operations which must be built into the expert system. One item a programmer expects is matching operations. If there is a push on a stack, one expects to find a pop operation elsewhere in the design. There is also a certain ordering of these operations. With a stack, the expectation is tbat a pusb will precede the pop, additionally if "empty" is included the empty test will come before a pop.

On a file data type an "open" will precede a "close" and if a design includes either one, then the other is expected. The same goes for "retrieve" and "store."

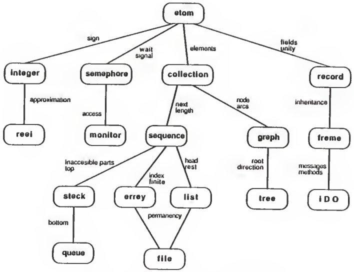
There are other combinations that are highly probable if the first element is present, but there is a comparable probability that the second element appears by itself. These operation pairs include "remove", "add" and "pred", "succ."

Awareness of such combinations will assist the expert system to detect potential missing pieces in software design.

4.4 Data Characteristics

The characteristics of a data structure are parts that any computer programmer can identify. These characteristics control the selection of a particular data structure for a design. For example a record is chosen over an array because the fields may be heterogeneous. The various attributes become important depending upon how the data objects are combined. Algorithms are simply the means programmers use to manipulate the characteristics of the data structure.

The top of a hierarchy of characteristics of data structures, Figure 4.2, is the atom. All data types are constructed from atoms. A characteristic of an atom is that each has a



Hierarchy uf data types by characteristics

Figure 4.2

value. The value is appropriate for that data type or it may be undefined. When designing a project, a designer must be aware of the range of values he wants represented by a variable and he must choose the data type that represents that range.

The characteristics of the other data structures identified as knowledge needed for an expert system are explained in Appendix B.

Chapter 5

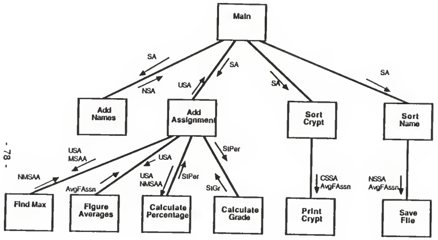
The Grader Program

5.1 Second Evaluation

To demonstrate that equable facts for analysis can be extracted from other ERL design specifications and that the rules for interpreting the facts are consistent, I will go through another example. This design is for a program to keep track of student grades for a class. In addition to demonstrating that the facts are consistent, this example has more slots of the Entity Relationship Level document filled than does the first example.

Figure 5.1 shows the hierarchy diagram of the design. Figure 5.2 is the ERL specification for the grader program. The purpose of each module in this example is understood by more people than the first example. Two reasons for this understanding are: The example is small and a program to calculate grades is familiar. I shall use the rules and variations of rules that have been previously discussed or rules that result from the additional information supplied in the ERL and I shall compare the results with the human understanding of the problem.

5.2 Evaluation



Hierarchy Diagram Grader Program

Figure 5.1

```
Activity        : Main
    comment     : Control section
    ext_input   : $Menu_selections
    input       : $Student_Array$
    output      : $Student_Array$
    calls       : Add_Names
                  Add_Assignment
                  Sort_Crypt
                  Sort_Name
    mode        : *Control*
                  *Insert_Name*
                  *Add_Grade*
                  *Print_Report*
                  *Save_Spreadsheet*


Activity        : Add_Names
    comment     : Adds a new student to a class or builds
                  original file.
    keywords    : insertion
    ext_input   : $Student_Name$
                  $Crypt_Name$
    input       : $Student_Array$
    output      : $New_Student_Array$
    mode        : *Insert_Name*
    assertion   : Length of $Student_Array$ is increased.


Activity        : Add_Assignment
    comment     : adds the score each student received to
                  the spreadsheet.
    keywords    : insertion
    ext_input   : $Student_Score$
    input       : $Student_Array$
    output      : $Updated_Student_Array$
    mode        : *Add_Grade*
    calls       : Find_Max
                  Figure_Average
                  Calculate_Percentage
                  Calculate_Grade
```

Figure 5.2
Grader Program ERL

```
Activity        : Find_Max
    comment     : finds the maximum grade of all the stu-
                  dents to use
                  as the maximum possibla score.
    keywords    : maximum
    input       : $Upoated_Student_Array$
                  $Max_Score_for_All_Assignments$
    output      : $New_Max_Score_for_All_Assignments$
    mode        : *Add_Grade*

Activity        : Figure_Averages
    comment     : calculatas the average for tha class for
                  this assignment.
    keywords    : average
    uses        : $Updated_Student_Array$
    output      : $Averaga_for_Assignment$
    mode        : *Add_Grade*

Activity        : Calculate_Percentage
    comment     : calculate the new percent score for each
                  student.
    keywords    : ratio
    input       : $Updated_Student_Array$
                : $New_Max_Scora_for_All_Assignments$
    output      : $Studants_Percentage$
    mode        : *Add_Grade*


Activity        : Calculate_Graoe
    comment     : caluclates tha lettar graoe from the
                  students percentage.
    keywords    : comparison
    input       : $Students_Percentaga$
    output      : $Students_Letter_Grade$
    mode        : *Add_Grade*


Activity        : Sort_Crypt
    comment     : sort the student array by crypt names
                  for print out.
    keywords    : sort
    input       : $Stuoent_Array$
    output      : $Crypt_Sorted_Stuoent_Array$
    mode        : *Sort_Crypt*
    calls       : Print_Crypt
```

Figure 5.2
Graoer Program ERL

```
Activity        : Print_Crypt
    comment     : prints the sorted crypt file for the
                  students
                  showing their grade.
    keywords    : output to printer
    input       : $Crypt_Sorted_Student_Arrays$
                  $Average_for_Assignments$
    ext_output  : $Printed_Student_Arrays$
                  $Printed_Average_for_Assignments$
    mode        : *Sort_Crypt*

Activity        : Sort_Name
    comment     : resorts the student array to name order
                  so it can be filed.
    keywords    : sort
    input       : $Student_Array$
    output      : $Name_Sorted_Student_Arrays$
    mode        : *Save_Spreadsheet*
    calls       : Save_File

Activity        : Save_File
    comment     : saves the student array on disk.
    keywords    : save to secondary storage
    input       : $Name_Sorted_Student_Arrays$
                  $Average_for_Assignments$
    ext_output  : $Filed_Student_Arrays$
                  $Filed_Average_for_Assignments$
    mode        : *Save_Spreadsheet*

Type            : $Student_Array$
media           : secondary storage
    structure   : array [1..120] of $Student_Records$

Type            : $New_Student_Arrays$
    structure   : array [1..120] of $Student_Records$

Type            : $Updated_Student_Arrays$
    structure   : array [1..120] of $Student_Records$

Type            : $Crypt_Sorted_Student_Arrays$
    structure   : array [1..120] of $Student_Records$

Tyde            : $Printed_Student_Arrays$
    media       : printer
    structure   : array [1..120] of $Student_Records$
```

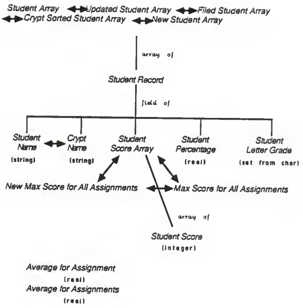
Figure 5.2
Grader Program ERL

```
Type             : $Filed_Studant_Array$
    media        : secondary storage
    structure    : array [1..120] of $Student_Records$

Type             : $Student_Records$
    structure    : $Student_Names$
                   $Crypt_Names$
                   $Student_Scores_Arrays$
                   $Students_Percentages$
                   $Students_Letter_Grades$

Type             : $Studant_Scores_Arrays$
    structure    : array [1..15] of $Student_Scores$

Input            : $Menu_salections$
    media        : crt
    structure    : a string from set (Add_Name,
                   Add_Assignment, Print, Save)

Input            : $Student_Names$
    media        : crt
    structure    : string of characters

Input            : $Crypt_Names$
    media        : crt
    structure    : string of alpnanumerics

Input            : $Student_Scores$
    media        : crt
    structure    : integer from 0..200

Output           : $Students_Percentagas$
    media        : printer and secondary storage
    structure    : real from 0.0 to 105.0

Output           : $Students_Letter_Grades$
    media        : crt, printer and secondary storage
    structure    : character from the set (A, B, C, D, F,
                   I, W)
```

Figure 5.2
Grader Program ERL

```
Input              : $Max_Score_for_All_Assignments$
    media          : secondary storage
    structure      : array [1..15] of $Student_Scores$

Output             : $New_Max_Score_for_All_Assignments$
    media          : secondary storage
    structure      : array [1..15] of $Student_Scores$

Output             : $Average_for_Assignments$
    media          : printer
    structure      : real from 0.0..100.0

Input_output       : $Average_for_Assignments$
    media          : printer and secondary storage
    structure      : real from 0.0..100.0


MODE_TABLE
    Modes          : *Control*
                     *Insert_Name*
                     *Add_Grade*
                     *Print_Report*
                     *Save_Spreadsheet*

    Initial_Mode          : *Control*

    Allowed_Mode_Transitions
       $Menu_selection$ = 'Add_name'       : *Control* ->
*Insert_Name*
       $Menu_selection$ = 'Add_assignment': *Control* ->
*Add_Grade*
       $Menu_selection$ = 'Print'          : *Control* ->
*Print_Report*
       $Menu_selection$ = 'Save'           : *Control* ->
*Save_Spreadsheet*
       $Student_Name$   = ''               : *Insert_Name* ->
*Control*
       Calculate_Grade                     : *Add_Grade* ->
*Control*
       Print_Crypt                         : *Print_Report ->
*Control*
       Save_File                           :
*Save_Spreadsheet -> *Control*
```

Figure 5.2
Grader Program ERL



Data Structures of Grader Program

Figure 5.3

One of the results of the previous analysis was that the first step in the study of a design is the identification of the relations among the data structures. This example has admirable data structures for a small project. The derived graph of the data structures is shown in Figure 5.3.

Important details about the graph are the number of data items that have identical structures and the construction of the data structures. An goal of the design analysis will be to discover the relation between Student_Scores, Max_Score_for_All_Assignments and New_Max_Score_for_All_Assignments because they have the same structure. The variations on Student_Array probably indicate that a change has occurred within the data. If a project design contains identically constructed data structures then the expert system will have an expectation created to find relationships between these structures. However, the same assumption cannot be made with the base types: -- real, integer, boolean, character and strings. Strings are not matched because two strings are rarely related.

5.2.1 Main

Analysis of the design starts with the Main module at the top of the hierarchy chart. As mentioned earlier, most designs will have two or more layers at the top which contain control activities. The conclusion that Main is a control module is supported by Main being at the top of the hierarchy and by the word "control" in the comment slot as well as by the name "Main."

The twist in this analysis is the external input of Menu_Selection. "Menu" is defined in Wehster as "a list shown on the display of a computer from which a user can select the operation the computer is to perform." The fact that the structure of Menu_Selection is one of a "set of strings" indicates that the intention of this module is still control, with the user being ahle to control the program. A note ahout this module would be sent to the designer suggesting a prompt as external output to the terminal because of the external input, Menu_Selection, from the keyboard.

The Menu_Selection input into the Main module has biased the analysis to expect the structure of a transaction program, whereas the example of chapter 2 was a transform program.

5.2.2 Add Name

The Add_Name activity yields these facts: the word "add" is contained in the name, the keyword is "insertion", the external inputs are files of the record which is an element of the array that is output, the array input has the same structure as the array output, and an input and output have the same name except for the word "new" concatenated to the front of the input name. The evidence says the intent of this activity is to build a data structure.

Although a count of the inputs and outputs does not reveal that the module is afferent, the details of the operation impart the knowledge that more data are being passed into the system than are passed from the system. Also included is the presence of two external inputs. Another message will be sent to the designer about the absence of prompts for keyboard inputs.

5.2.3 Add Assignment

Evidence collected from the ERL on the activity Add_Assignment includes the external inputs of Student_Score and the input of Student_Array. The output of Updated_Student_Array declares that information in the structure of Student_Array will be modified. Justification of the modification comes from the fact that the output has the same structure as an input and from the considerable matching of names. The difference in the names also contributes to the idea of modification. Words such as "update", "new", "revised" and "edited" imply change. Again the keyword "insertion" indicates that some type of addition is to be made.

A problem exists with the module concerning matching of data abstraction levels as discussed earlier. The inputs are at different levels of complexity. Student_Score is an entry that is an element of an array that is a field of the Student_Record of which the Student_Array is built. Since a perceived intent of this module is to add a score into the Student_Record, the operation "Add_Assignment" should be taken down one more level in the hierarchy, and the Student_Record rather than the Student_Array should be passed to it.

Another way of evaluating how to match the data abstractions is that the lower level module should build the Student_Scores_Array field and have that data structure added to the Student_Record and then to Student_Array. In either case the analysis of the module results in messages to the designer indicating the mismatch of data abstraction and again the absence of prompts for keyboard inputs.

5.2.4 Sort Crypt

In the Sort_Crypt module, the input is the Student_Array and the output is Crypt_Sorted_Student_Array. There is still a common base in the names used and the structures are the same. The keyword "sort" is an operation one recognizes as being performed on sequences, (e.g. arrays). A search of all the data types gives no indication of stacks or queues, so a sort is a reasonable operation for this module.

A sort is a rearranging of elements. Therefore, the Student_Records are to be rearranged by some criteria. Since records contain fields it is presumable that the sort will key on one of the fields. A human designer, even though he has not been told on what field the sort must be based on, would be able to identify that field by the correspondence of the word "crypt" in the name of the module and the Crypt_Name field of the Student_Record. An expert system should be able to deduce the same information. A sort is a transform operation and the module is so tagged.

5.2.5 Sort Name

The analysis of the Sort_Name module proceeds along the same lines as Sort_Crypt and determines that Sort_Name is a transform module, although it is at the right end of the second level of the hierarchy. When attempting to determine which field to sort on, one finds two fields in Student_Record that have "Name" as part of their title. There are several clues in the way the data is constructed that will assist in resolving which field is the sort field. The first hint is the order. Student_Name is the more probable choice for the sort field because it is the first field. The fact that Student_Name is a field of the Student_Record that is part of Student_Array points to the word "Student" as being a primary key of the records in this array. Supporting evidence is also generated from the observation that there appears to be a sort module that keys on the other field with "Name" in it. Good designs do not have multiple modules that perform identical operations, especially if the modules are called from the same parent module.

Now that all the modules across the second level have been examined, a look across the breadth of this level evaluates how well the modules link together. Examination of the inputs and outputs indicates that all modules operate on the Student_Array. It appears as if each module modifies the data in the Student_Array in one way or another. Two modules cause concern in the first level; Sort_Crypt and Sort_Name. An alert signal is triggered when two sorts are seen together. The system should be analyzed to see if one may be eliminated either by a redesign of the system, or by a change of data structures. Also, the system expects to detect efferent modules in the second level of a design. Because no efferent modules were detected, a message will be issued describing this fact.

A sort is a permutation of the array without alteration of the information contained in the elements. But because sorts are known to consume large portions of computing time it would be beneficial to try to eliminate one of them. The control flow, which is coordinated from the Main with a menu, does not appear to give any order to the data flow, which may mean that the sorts must stay as they are. However, the system should question the designer for the need of two sorts, suggesting a rearrangement of how modules are called. If a sort cannot be eliminated, perhaps it may be moved so that it is not performed as often.

The analysis of the Add_Assignment module is hindered because analysis already indicates the need for some changes. The system will also note that not all fields of Student_Record have had assignments made to them.

5.2.6 Find Max

Proceeding to the next layer of the hierarchy the analysis will build on the information already extracted. In the Find_Max module, the inputs include Updated_Student_Array and Max_Score_for_All_Assignments. The output is New_Max_Score_for_All_Assignments. Keyword is maximum. A designer will be able to take this information and deduce that the purpose of this module is to find the maximum score in the Student_Scores_Array field in the column of the assignment being added. Several inferences, with a complex set of interactions, are made to arrive at this conclusion. "Maximum" is an operation that makes sense on collections, specifically on heterogeneous collections. Therefore, one would expect some kind of looping through the Student_Records of the Student_Array to determine a maximum. This is a transform module because all the data to be operated on are already in the machine. The inputs to the module point to the Student_Scores_Array field in the Student_Record as the field for which the maximum is determined. The Student_Scores_Array choice is further supported by a match in the number of elements in that array with the number of elements in the Max_Score_for_All_Assignments array. Additionally because the parent module deals with the Student_Scores_Array, the Find_Max module will probably do the same.

Yet, a person's concept of a maximum routine is to return a value, not an array of values as Max_Scores_for_All_Assignments would indicate. A person evaluating the Find_Max module might guess that the Max_Score_for_All_Assignments is to contain the maximum for each column in the array. A rule can be developed to evaluate designs along the same lines. However, I believe that the rule to cover this situation may not be consistent across all designs. Further study is needed to determine what is the most common error which produces a set of facts similar to those present in this module. The word "all" in the Max_Score_for_All_Assignments brings cognitions that somehow a maximum is needed for the entire Student_ Scores_Array, not just for a column or a row.

A human understands that there is a limiting factor on the array. The expert system needs to know that an index can single out an element in an array. It also knows that the concept of maximum is to return a single value. The expert system may find it difficult to discover that a single value is returned but that the value must be appended to the array Max_Score_for_All_Assignments. A way to check this hypothesis would be to return to the parent module and see if a limit to the array would help resolve the problem in Add_Assignment. The index could fit into the Add_Assignment module, and it would further emphasize the need for a separate module to be created one level down in the hierarchy.

There is a gap in the ability of the expert system to determine the purpose of this module. Earlier I discussed the value of being able to pull out words from the names of modules and data items. A mechanism is needed to make the word "assignment" synonymous with the concept of one additional score for each student. The machine would then be able to evaluate Find_Max and Add_Assignment on par with a human.

5.2.7 Figure Averages

The analysis of the Figure_Averages module will proceed along identical lines as the Find_Max, but with an additional restriction. Whereas one can rationalize about finding a maximum string, calculating an average string does not have any standard meaning. The average operation deals with a collection of numbers. The output, Average_for_Assignment, is of an appropriate type, and is only one value, so it makes sense. The analysis discovered that the calling module manipulates the Student_Scores_Array field of the Student_Record. This will guide the input analysis to use the same field in the Figure_Averages module. Because the Student_Scores_Array is one of only two numeric fields in the Student_Record, the expert system will be confident of that selection. The system sees no problem in averaging a two-dimensional array which is what the input data structure appears to be at this point. However, the system also knows about averaging one-dimensional arrays. Because the system recalls the problem of analyzing the calling module and the other module at this level, a question mark about the correctness of this module design is suspect. A message will be sent to the user detailing the two choices. "Average" has the implication that all the data needed is already present, therefore this activity will be marked as transform.

5.2.8 Calculate Percentage

Evaluation of the Calculate_Percentage module begins with the inputs and outputs. The "uses" label or slot for Updated_ Student_Array and New_Max_Score_for_All_Assignments means that these data are needed by the module for some calculation, but they are not changed by the calculation. The name of the module has two words to key on, "calculate" and "percentage." The keyword is "ratio." "Calculate" is too ambiguous to be used by the machine except to help label the activity as transform. Calculations run from simple to complex. "Percentage" is defined in Webster as "a part of a whole expressed in hundredths" and is very helpful in interpreting the intent of this module. The keyword "ratio" matches well with the definition of percentage, i.e. "part of a whole", therefore the expert system will approve this module. The inputs to this module are examined to check that there are numbers involved, so after some calculations a division can be made, and that the output is numeric. The expert system being described must only evaluate the design, is does not generate code. Therefore, the system is concerned not with how the exact calculation is done, but with which inputs seem reasonable so that a percentage could be generated for output.

5.2.9 Calculate Grade

The Calculate_Grade module is examined in the same manner as the Calculate_Percentage activity. The difference between the two modules is that a single input is given and a single output is produced. Since the input and output of this module are of different types, some type of mapping function is indicated. Further support is provided for this conclusion by the keyword "comparison." The expert system knows that mapping functions exist between different data types and that the modules are generally transform modules. The system can reason that the mapping will be explicitly developed in the code, but to make this design document more useful the information about the mapping from the requirements phase should be transferred into this document. Part of verifying a design is to insure that all the requirements are specified; therefore, I introduced the intrinsic knowledge slot to have a location to place this type of information.

The type of information included in the intrinsic knowledge slot is decision-specific and not necessarily machine translatable. The automated system will simply check for the presence of the intrinsic knowledge field. The information included in the field should be sufficient so that decision code could be written.

With completion of the analysis of the set of modules called from the same parent, the group as a whole must be analyzed. Using this method, one first examines the inputs and outputs. Three modules, Find_Max, Figure_Averages and Calculate_Percentage, use the Updated_Student_Array. The data flow indicates that some change occurs to the Student_Array in the Add_Assingment module before it is passed on to these modules. The last module, Calculate_Grade has the input of Student_Percentage which is the output of Calculate_Percentage. Because of the close coupling of Calculate_Grade to Calculate_Percentage and the non-use of the Updated_Student_Array by the Calculate_Grade module, a suggestion will be made to the user to move the Calculate_Grade module to be called from the Calculate_Percentage module.

The output from Find_Max is used by Calculate_Percentage, which signifies sequential operation between the two modules. Yet there is no connection with the output of Figure_Averages or with the other modules in this branch. If the designer had changed the slot of Updated_Student_Array in Find_Max to "uses" instead of "input", which indicates a change occurring, the possibility for concurrency between these two modules would be pointed out.

The output from Calculate_Percentage and Calculate_Grade bestows a different meaning on the Add_Assignment module. The only field of Student_Record pointed to before the examination of Calculate_Percentage and Calculate_Grade, was the Student_ Scores_Array. Analysis now has two additional fields of Student_Record referenced: Students_Percentage and Students_ Letter_Grade.

The system has marked all the fields of Student_Record as being assigned. A view of the design now reveals one first level module that inserts a new Student_Record into Student_ Array only filling in fields it has identified as keys, Student_Name and Crypt_Name. The system also sees that the modules currently under analysis, Add_Assignment's called processes, create a new record with all the fields assigned except for those identified as key fields. An analyzer of the design, human or computer, knows there are no more modules in this section for analysis, and it knows that adding a new record without including the key information does not make sense. Consequently, the two proposals for the Add_Assignment module will be reviewed.

The expert system already evaluated the words "add" and "insertion" in the Add_Assignment module and experienced difficulty in the levels of abstraction between Student_Score and Student_Array. With the added information of doubting that the purpose of the module is to insert a new record into the Student_Array, a single interpretation of the intent of the module remains. The system will convey this information to the designer along with the explanations of the ambiguities encountered.

5.2.10 Print Crypt

During the evaluation of the Print_Crypt module the system discovers that the inputs and outputs harmonize well with the keyword "output to printer" and the "print" word in the activity name. The outputs of this module are external to the program and they have the same structure as the inputs. This labels the function of this module as an efferent activity of the program.

5.2.11 Save File

The last module examined in this design is the Save_File module. It has several inputs, all of which have been generated earlier, and the outputs all correspond with the "File" part of the name. Therefore, this module passes inspection and will be labeled as an efferent module. This module is the only called activity of Sort_Name, so there is nothing to evaluate in examining all the called processes. The results are passed back to Sort_Name to affect the certainity factor of its evaluation.

5.2.12 Review

All modules and all relevant combinations of modules have been examined. The designers have been made aware of questionable areas in the Add_Assignment section. A look at the overall design will check if everything matches with expectations.

When the expert system analyzed the Save_File module, an expectation was created that a corresponding Retrieve_File activity would exist in the design. When the expert system encountered modules that had as inputs Student_Array, which is declared as input from secondary storage in the ERL, it figured each module read the array in from the file. This matches the idea of menu control; If you make some changes you don't want to keep, you don't save the file, and you can start with the same data on each selection of the menu.

The system would also note that the Save_File module uses inputs not only of the type Student_Array but also of type Max_Score_for_All_Assignments and Average_for_Assignment. This indicates that the three different items were all saved together. Nowhere in the design are they read together. Additionally no module declared Max_Score_for_All_Assignments as an external input, which is indicated by its declaration as "Input" in the type entities of the ERL. Although the three entities may each be saved on a different file, the expert system will print a message relating the fact that these data are not all read together.

By no means have all the cases been exposed with the two examples discussed in this paper, but a start has been made to determine the information needed to develop an expert system.

Chapter 6

Results and Conclusions

A set of rules has been generated that given a software design presented in the form of an Entity - Relationship - Level document, an expert system can determine a generalized concept of the intent of most activities in the design. With the information of the inputs, outputs and the deduced function of a module, the design can be compared to a model and discrepancies can be pointed out to the designer. The knowledge used to analyze a design comes from three major sources which are known to software designers and can be coded into an expert system.

First, people use meaningful names in designs. Although an argument can be made that interpreting the names used involves natural language understanding, I believe the domain is sufficiently restricted that keyword recognition can be used. Weizenbaum's ELIZA uses similar keyword recognition. I've demonstrated how the semantics of words useful to analyzing design can be extracted and used in a manner that can be encoded into an expert system.

Second, the structure of the inputs and outputs of an activity can reveal the generalized purpose of the activity. If the data structures are related (e.g. through an intermediate data type in the ERL) then the module may be considered as constructing a data structure or extracting information from a more complex data structure. If the inputs and outputs are not related through the data structures then a mapping from the input to the output is suggested.

Third, the correlation of data structures with the deduced functions results in the creation of other expectations. These expectations of a module are based on prototype program plans and the expert system either confirms the expectations or generates a query to the user. Plans often involve operations expecting certain data types as inputs and producing other specific data types. These programming plans serve as the foundation for the analysis.

The knowledge needed to analyze a design is rather specific. It may appear that a considerable amount of "common sense" knowledge is needed, but the domain is restricted, limiting ambiguous interpretations of most of the words used in software design. The narrow domain also means that the number of rules needed to reason "intelligently" is probably not overwhelming.

I do not know if the analysis of the example in chapter 2 pointed to any potential trouble spots or not. The design of the grader program of chapter 5 was changed as a result of the analysis. Some of the recommendations from the analysis were followed, but many of the design changes reflected the designer's own concepts after certain questions were presented.

The system discussed in this paper will only question a designer about suspected areas. It cannot determine if the design is good or bad, since there is no general agreement among humans. Design is and remains an art form and the designer is not forced to produce designs according to rigid algorithms.

The use of the Entity - Relationship - Level document needs to be made consistent. The several different interpretations of how to name inputs and outputs to show change inside a module will have to be made consistent for all users of the system. Other features such as declaring inputs or outputs external to a module, recently added to the ERL model, must be firmly and consistently used by all users if the expert system is to be successful.

My recommendation for use of the ERL is: The term "uses" is for specifying that an input is used but that it is not altered by an activity. If the output to the calling module is different than the input passed to the activity then use the "input" and "output" slots with the same variable name. If the activity changes an input before passing it as an output to its children processes then the output to the children should have a different name. The same name of the variable in the output slot signifies that the data is returned to the parent.

The knowledge required to start building an expert system has been extracted from the two examples in this paper. The system is designed to grow. As more examples are examined and as new rules are discovered, they can be added to the expert system. Rules developed from the first example were used in the second example and the new cases not seen before resulted in the addition of new rules to the system. Although two examples are not enough to set a pattern, I believe that to incorporate new types of designs into the expert system will not require a new set of rules. Thus, I anticipate that the number of rules to analyze a larger set of designs will level off as occurred in PECOS [Ba85]. This system should be able to point out activities that are incomplete or inconsistent. With potential errors revealed, the designer can concentrate on those areas, producing a design that makes more intuitive sense.

Chapter 7

Future Work

The future work to convert the expert system described into a useful product involves more than just implementing the system.

I was the sole source of interpretation of the designs. To validate the facts extracted and the rules generated in this paper, other persons familiar with the ERL and the Yourdon - Constantine model need to be observed as they evaluate designs. The different techniques, facts, rules and determinations they use will be compared and contrasted with those described here to develop a more universal set of knowledge.

I feel it is necessary to analyze software designs from industry. Industrial sources should be able to provide larger designs that are produced by more experienced programmers and that have a consistent level of detail.

Of course this system must be implemented and tested. I expect to use two artificial intelligence paradigms, rule based and object oriented programming. The two entities of the ERL discussed in this paper, data and activities, fit into hierarchies. Object oriented programming is well suited to this facet of the problem structure. Facts and rules was the other approach used in analyzing the modules of the design, therefore, the rule based approach resembles the technique used by humans.

Finally the results of the implementation must be evaluated to discern the effectiveness of the system in detecting potential trouble spots. I hope that the automated analysis of software designs will point out difficult areas of a project early in the software lifecycle so that changes to the design can be made earlier at less cost. Several iterations are expected to be necessary to optimize the automated analysis for industrial use.

Bibliography

[Ab85] Abelson, H., G.J. Sussman and J. Sussman. Structure and Interpretation of Computer Programs. Cambridge, MA. MIT Press; (1985).

[Ad85] Adelson, Beth and Elliot Soloway. "The Role of Domain Experience in Software Design." IEEE Transactions on Software Engineering. Vol. SE-11; (Nov. 1985) 1351-1360.

[Ah74] Aho, Alfred, John Hopcroft, and Jeffrey Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley; Reading, Mass. (1974).

[Ba79] Barstow, David. "An Experiment in Knowledge-based Automatic Programming." Artificial Intelligence. Vol. 12; (1979) 79-119.

[Ba85] Barstow, David. "On Convergence Toward a Database of Program Transformation." ACM Transactions on Programming Languages and Systems Vol 7; (Jan. 1985), 1-9.

[Bh84] Boehm, Barry W. "Verifying and Validating Software Requirements and Design Specifications." IEEE Software. (Jan. 1984) 75-88.

[Bn84] Bonet, R. and A. Kung. "Structuring into Subsystems: The Experience of a Prototyping Approach." SIGSOFT Software Engineering Notes. Vol. 9; (Oct. 1984) 23-27.

[Bo85] Borgida, A., S. Greenspan and J. Mylopoulos. "Knowledge Representation as the Basis for Requirements Specifications." Computer. Vol. 18; (April 1985) 82-91.

[Bo84] Borgida, A., J. Mylopoulos, and H.K.T. Wong. "Generalization as a Basis for Software Specification." On Conceptual Modeling: Perspectives from Artificial Intelligence, Database and Programming Languages. M. Brodie, J. Mylopoulos, & J. Schmidt eds. Springer Verley, New York (1984) 87-114.

[Br85] Braegger, Richard P., Andreas M. Dudler, Jrerg Rebsamen and Carl August Zehnder. "Gambit: An Interactive Database Design Tool for Data Structures, Integrity Constraints, and Transactions." IEEE Transactions on Software Engineering Vol. SE-11; (July 1985) 574-583.

[Br75] Brinch Hansen, Per. "The Programming Language Concurrent Pascal" IEEE Transactions on Software Engineering Vol. SE-1; (June 1975) 199-207.

[Dm82] Demetrovics, J., E. Knuth, and P. Rado. "Specification Meta Systems" Computer. Vol. 18; (May 1982), 29-35.

[DP83] DePree, Robert W. "Pattern Recognition in Software Engineering." IEEE Computer (1983) 48-53.

[De84] Deitel, Harvey M. An Introduction to Operating Systems. Addison Wesley; Reading, Mass (1984).

[DM84] DeMarco, Tom. "An Algorithm for Sizing Software Products." ACM Sigmetrics Vol. 12; (Spring-Summer 1984) 13-22.

[Do84] Domeshek, E.A., D.G. Shapiro, J.S. Dean, and B.P. McCune. "Informal Study of Program Comprehension." Advanced Information & Decision Systems, AD-A142224/5. (March 1984).

[Eh84] Ehrlich, Kate and Elliot Soloway. "An Empirical Investigation of the Tacit Plan Knowledge in Programming." Human Factors in Computing Systems. J.C. Thomas & M.L. Schneider (Eds.) Ablex Publishing Corp.; Norwood, NJ. (1984) 113-133.

[Fo84] Foster, C.C., V. Rao, J.A. Wills, C. Bron, E. Soloway, J. Bonar and K. Ehrlich. "Cognitive Strategies and Looping Constructs: An Empirical Study." Communications of the ACM Vol. 27; (Oct 1984) 1048-1051.

[Go83] Gonnet G.H. and F. W. Tompa. "A Construction Approach to the Design of Algorithms and Their Data Structures." Communications of the ACM. Vol. 26; (November 1983) 912-920.

[Gr82a] Greenspan, Sol, John Mylopoulos, and Alex Borgida. "Principles for Requirements and Design Languages: The TAXIS Project." Requirement Engineering Environments. Proceedings of International Symposium on Current Issues. North Holland, Inc.; New York (1982) 107-113.

[Gr82b] Greenspan, Sol, John Mylopoulos, and Alex Borgida. "Capturing More World Knowledge in the Requirements Specification." 6th International Conference on Software Engineering. (1982) 225-234.

[Gs83] Grosch, Josef. "Type Derivation Graphs -- A Way to Visualize the Type Building Possibilities of Programming Languages." SIGPLAN Notices. Vol. 18; (December 1983) 60-68.

[Gu84] Gustafson, David A. "A Requirement Model for the 5th Generation." Proceedings of 1984 ACM Annual Conference (Oct. 1984) 149-156.

[Gu77] Guttag, John. "Abstract Data Type & Development of Data Structures." Communications of the ACM. Vol. 20; (June 1977) 396-404.

[Ha83] Haynes-Roth, Waterman, and Lenat (Eds.) Building Expert Systems Addison Wesley; Reading, Mass. (1983).

[Jo85] Johnson, W. Lewis and Elliot Soloway. "PROUST" Byte (April 1985) 179-190.

[Kn73] Knuth, Donald. Sorting & Searching: The Art of Computer Programming. Addison Wesley; Reading, Mass. (1973).

[Ls83] Lescanne, P. "Behavioural Categoricity of Abstract Data Type Specification." Computer Journal. Vol. 26; (November 1983) 289-292.

[Li77] Liskov, B. H. and S. Ziller. "Specification Techniques for Data Abstraction." Proceedings of the International Conference on Reliable Software. SIGPLAN Notices. Vol. 12; (1977).

[Ma83] Matskin, M. B. "Debugging Tools for a System with Automatic Program Synthesis" Programming & Computer Software. Vol 9; (July-Aug 1983), 173-177.

[Ma84] Matsumoto, Yoshihiro. "Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels." IEEE Transaction on Software Engineering. Vol. SE-10; (September 1984) 502-513.

[Mc84] McCabe, T. J. Specification Complexity and Validation. AD-A141419/2,4 (May 1984).

[Ri81] Rich, Charles. "Inspection Methods in Programming." AI-TR-604. Artificial Intelligence Laboratory. MIT (1981).

[Sh84] Shapiro, Daniel G., Jeffrey S Dean, and Brian P McCune. "A Knowledge Base for Supporting an Intelligent Program Editor." 7th International Conference of Software Engineering (March 1984) 381-386.

[Sh83] Shapiro, Daniel G., and Brian P. McCune. "The Intelligent Program Editor: A Knowledge Based System for Supporting Program and Documentation Maintenance." Proceedings of the Trends and Applications Conference of the IEEE (May 1983) p226-232.

[So84] Soloway, Elliot and Kate Ehrlich. "Empirical Studies of Programming Knowledge." IEEE Transactions on Software Engineering. Vol. SE-10; (September 1984) 595-609.

[Sw84] Sowa, John. Conceptual Structures: Information Processing in Mind & Machine. Addison Wesley; Reading, Mass (1984).

[St84a] Stubbs, M. "Resolution of Structure Clashes by Structure Inversion." Computer Bulletin Vol. 2; (March 1984) 8-19.

[St84b] Stubbs, M. "An Examination of the Resolution of Structure Clashes by Structure Inversion." Computer Journal. Vol. 27; (Nov 1984) p354-361.

[Tr84] Tremblay, Jean-Paul and Paul G Sorenson. An Introduction to Data Structures with Applications. McGraw-Hill; New York, NY (1984).

[Un86] Unger, E. A. "Intelligent Data Objects: A Concept Useful in Networks." Kansas State University. (1986).

[Wa78] Waters, Richard C. "Automatic Analysis of the Logical Structure of Programs." AI-TR-492. Artificial Intelligence Labortory. MIT (1978).

[Wa82] Waters, Richard C. "The Programmer's Apprentice: Knowledge Based Program Editing." IEEE Transactions on Software Engineering. Vol. SE-8; (Jan. 1982) 1-12.

[Wa85] Waters, Richard C. "The Programmer's Apprentice: A Session with KBEmacs." IEEE Transactions on Software Engineering. Vol. SE-11; (Nov. 1985) 1296-1320.

[Wr83] Wirsing, M., P. Pepper, H. Partsch, and W. Dosch. "On Hierarchies of Abstract Data Types." Acta Informatica Vol. 20; (1983) 1-33.

[Wi76] Wirth, Nichlaus. Algorithm + Data Structure = Program. Prentice_Hall, Inc. Englewood Cliffs, NJ. (1976).

[Yo79] Yourdon, Edward & Larry L. Constantine Structured Design: Fundamentals of a Discipline of Computer Program and System Design. Prentice-Hall, Inc. Englewood Cliffs, NJ. (1979).

[Ze84] Zelkowitz, M. V. "A Taxonomy of Prototype Designs." SIGSOFT Software Engineering Notices Vol. 9; (Oct. 1984), 11-12.

Appendix A

Data Structure Operations

To determine a viewpoint on how to look at data structures in assisting the analysis of designs reminds me of the question of which came first, the chicken or the egg. Do data structures define operations allowed on them or do the operations define the data structure? For the computer, all data are a voltage or no voltage. The data type places attributes on the bits specifying the operation to interpert the bits in different ways.

The base data types, integer, character, boolean and real can have values assigned to them, can have their contents retrieved or two values can be compared. In most imperative programming languages all but the boolean can be directly inputted or outputted and there are several math operations that can be performed on the numeric types.

The operations on records include: Modify, which is the concept of assigning a new value to a field without changing other portions of the record. This concept is different from assignment to an atom because with an atom the entire value is new, whereas only a portion of the record is new. An analogy is: putting new tires on a car, it is the same car, only with new components. Key-retrieve operation on a record is the concept that you can gain access to data contained in a record by knowing the identifying key to that record and matching the keys. This operation is identified with records but it is not truly useful unless there is some collection of records to apply the search to.

Frames are similar to records in the aspect that frames contain attributes in respective slots associated with a single object. However, in order for a data type to be commonly called a frame, there is an associated hierarchy of inheritance and relations between frames of different objects. The demons associated with a frame are operations that the user defines to be triggered when frame values are added, deleted or modified.

Similar to frames are items referred to as Intelligent Data Objects [Un86]. This abstract data type not only includes the operations associated with it, but may have information as to who can access the object, and when certain operations are allowed. An additional operation connected with an intelligent data object is a message. Messages can be sent requesting an action to take place, like to modify, or to request information given only a key.

Operations connected with a stack and a queue are push, pop and empty. There may be the need to break pop into an operation called top, to return the top value in the stack or queue, and pop to remove the top value from the structure.

Also depending upon the stack or queue implementation it may be necessary to include a full operation and a depth operation. These cover the operations that may be performed on a stack or queue. The terminology used is more common with stacks, but a simple mapping to terms computer scientists are more likely to use with queue is push - add, pop - remove, top - front and depth - length. The analyzer of a software design needs to know about all these basic operations. Additionally the expert system needs to be flexible to fit the applications of different software development shops.

If a design calls for a collection of objects then many more operations become important in making use of the computer. It now becomes necessary to be able to add elements to and remove elements from a collection. Other operations on collections include testing if the collection is empty, or if an element is contained as a member of a collection or if one collection is a subset of another. The operational concept of restrict is to derive a subset of a collection containing elements that maintain a certain property. The mathematical operations on sets that can be applied at this level include: union, intersection, and set difference.

A large group of operations is exposed when a designer thinks about sequences. He can write modules that will return the length of a sequence, search a sequence for a particular value, map the sequence to some other sequence, can truncate a sequence after certain criteria are met and return the next element in the sequence. The operation truncate is different from the operation restrict on a collection for the reason that the criterion for ending the sequence is based on the location in the sequence and not on the value of the element, as is the only way to restrict in a collection. A sequence may be truncated after the first 10 elements, whereas a subset may be restricted to elements whose value is less than 10.

An important subset of sequence in programming is the finite sequence. If a sequence is finite in size, the concept of sorting is valid. You can have the notion of searching an infinite sequence, but there is no intuition of sorting an infinite sequence.

If you are dealing with arrays, the indexing operation that allows random access into the array is important to understand how algorithms and data structures cooperate. The other operation concerning arrays is overflow, since an array in many languages must be of a predetermined size, you may have more information to insert in the array than the array can hold.

A list is an implementation of a sequence that does not necessarily have to be thought of in the context of being finite. The additional operations associated with a list data type beyond a sequence are head, which returns the first element of a list; cdr, which returns all the elements of a list except the first; and last, which returns the last element of the list.

A file has associated with it the create, open, close, store and retrieve operations which are connected with the permanency property of a file. Store and retrieve are distinct from input and output inasmuch as they carry the impression of reuse of the data object at a later time.

Because a next operation on a graph can return a set of elements, instead of a single element, there is a distinction on graphs to convey this concept. The successor operation conveys the thought that a set of elements is returned instead of just a single element. A designer of a software module dealing with graphs is likely to traverse the graph rather than search. The determination of a node being an end node is done with a terminal operation.

A tree is a specialized case of a graph that has operations to return the root, to return the parent of a node and to balance a tree.

The data structures used in concurrent computing are added to the taxonomy of data structures by noting the operations that distinguish the data types from each other. A semaphore has the operations signal and wait. A monitor has its monitor entry, similar to an intelligent data objects message.

A feature of many of the newer languages is an abstract data type. The abstract data type is a powerful concept for the user because he or she defines not only the properties of the abstract data type, but also define all the operations allowed on the data type. This is closely related to objects and concurrent processes. The concept is to totally encapsulate the data to prevent unknown side effects.

Appendix B

Data Structure Characteristics

The characteristics of a data structure are the nouns people use to describe the parts of the data types.

The numbers are branched off from atoms because they have a characteristic of sign. Although there may be little difference in comparing a number with the value 100 or the value 0, many operations are legal with positive numbers that are not legal with negative numbers. The distinction is large enough that certain design decisions are influenced by a number's sign. Other characteristics of a number's representation in a computer program are its range and size. If exact values are needed, then a designer is restricted to integers, however, if the design calls for a large range of numeric values and approximations can be handled, then reals are an acceptable data type.

The other short path through this hierarchy, as shown in Figure 4.2, are records which are composed of fields. Each field may have a different data type with its own set of values. Yet the fields of a record are descriptions of a single item. It is the desire to group information about single items into one structure that makes the record data type useful to a program designer.

A characteristic about a collection is that it is made up of an aggregation of elements. A designer of a system is concerned that the elements are all of the same type, but when he is thinking abstractly about the collection, he is not concerned with the type of elements whether they are atoms, records or other collections. This aggregation is generally thought of as a group of similar objects and not as different features of the same object as a record is thought of. The distinction that makes looping through an array meaningful is that it is different objects of the same type. Whereas looping through a record is enigmatic because it is different aspects of the same object.

A sequence has the characteristic attribute of a set with the additional properties of next and length. To a designer of a software system, it is those two properties that give sequences meaning. Wherever you are in a sequence there is one element that is the next element, even if it is a special end case. The other aspect of sequences a designer is concerned with is the sequence length. Even if the sequence is infinite, the program must be designed to handle the length characteristic.

One way to implement a sequence is with a list. A list inherits all the properties of a sequence. As a designer simulates execution of a program, he is generally concerned with two parts of the list, the head, or first element and the rest of the list, or to borrow a term from LISP, the cdr. The tail may not be a characteristic of the list structure because of the possibility of a circular list. A characteristic of a list is a pointer to the next element in the list.

Another way to implement a sequence is with an array. Two characteristics about arrays that designers must concern themselves with is the fact that an array is of finite length, and the other is the need for an index into the array. The length of an array may have to be determined at compile time, or can be established at run time depending upon the language. The index allows for random access into the array which may make it an attractive alternative to a list.

A comparison between an array and a list may help clarify the concept of the finite characteristic. While abstractly thinking about a project using an array you are conscious of the fact that there is some size which you cannot exceed, whereas with a list, there is no constraint on the size of the list. This is not to say that the size of a list is of no concern to a designer, but that the concern is handled differently.

A file is recognized as a data object that combines the characteristics of both arrays and lists. The implementation of a file, i.e. whether or not random access is possible, determines which type a file more closely models. An additional characteristic about a file is its permanency. The concept that the data will continue to exist beyond the life of the program is the useful aspect of a file.

An interesting branch from sequences is the data structures of stack and queue. A major component of these two data objects is the inaccessible part of the structure. The fact that they keep elements from being accessed is a key concept in their use. The stack access is only through the top, both for adding and deleting items. Since the location for adding an item is defined and the location where the next item comming from a stack is known, the sequence concept of next is applicable. The same is true for a queue with the additional concept of bottom for adding items to the queue and removing them only from the top.

When dealing with graphs, there are two component characteristics, nodes which generally denote objects and arcs which generally represent relations. The arcs are pointer to the next set of nodes. A more specialized form of a graph is a tree. A tree has two characteristics that identify it as a separate data structure. The first is a unique node from which you can follow various arcs to get to any other single node in the tree, this is known as the root. The other property that all arcs have is a direction that is away from the root node.

The combination of the operations and characteristics of a data type uniquely identify it among the computer science community. An expert system must have the knowledge about the characteristics and operations of common data structures in order that it can successfully analyze designs.

A THOUGHT EXPERIMENT TO DETERMINE THE KNOWLEDGE REQUIREMENTS OF AN EXPERT SYSTEM TO ANALYZE YOURDON - CONSTANTINE DESIGN HIERARCHIES

by

RICHARD E. COURTNEY

B. S., Southwestern College, 1976

AN ABSTRACT OF A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1986

Before an expert system can be implemented, the knowledge that is required to perform the task must be known. A thought experiment is described in which two example designs are evaluated. The questionable points of the designs are extracted and pointed out to the user. The extraction process, executed by a human, is broken down into facts and rules that could be coded into an expert system.

This paper demonstrates how the intent of an activity can be inferred from the name of the activity, the inputs to the activity, and the outputs from the activity. The information presented about the design is in an Entity - Relationship - Level (ERL) document as would be produced at the end of the Design Phase of the Software Lifecycle. The data obtained from the ERL is compared to a structural design model and a database of knowledge regarding data structures.

One area of knowledge common to software designers and programmers is abstract data structures. Programmers are cognizant of the operations allowed and not allowed on each of the various data structures. The knowledge base of data structures needed in the expert system is discussed in terms of two hierarchies: data characteristics and operations on abstract data types. The restrictions and expectations of operations in relation to each data structure is compared to the information in the ERL. This comparison provides sufficient data to evaluate the design specification for completeness, workability and understandability.